Raissa Corrêa Xavier de Almeida

Licenciada em Engenharia Informática

Divisão Mínima em Grafos

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientadora: Margarida Mamede, Professora Auxiliar,
Universidade NOVA de Lisboa

Júri

Presidente: Doutor Francisco de Moura e Castro Ascensão de Azevedo
Vogais: Doutor João Pedro Guerreiro Neto
Doutora Margarida Paula Neves Mamede

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2018

Divisão Mínima em Grafos

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

À minha família

Agradecimentos

Agradeço à minha orientadora, professora Margarida Mamede, que me deu a oportunidade de trabalhar neste projeto, além da ajuda durante o desenvolvimento do mesmo. Agredeço à Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, ao departamento de Informática e a todos os professores com os quais tive a oportunidade de ter aulas e aprender nestes últimos anos.

Gostaria de fazer um agradecimento especial ao Ricardo Carvalho Martins por ter cedido as implementações dos algoritmos propostos em sua tese. Também aos meus amigos, que estiveram sempre presentes durante o curso e especialmente durante a elaboração da dissertação.

E por último, gostaria de agradecer à minha família, que está sempre presente e pronta para apoiar as minhas decisões.

Logic is the foundation of the certainty of all the knowledge we acquire.

— Leonard Euler

Resumo

Com o crescimento dos dados armazenados nos *data centers* que estão localizados em diversas partes do mundo, surgiu o problema de onde as operações deviam ser executadas de modo a que o tráfego de dados entre *data centers* seja reduzido. Este problema pode ser interpretado como um corte-k mínimo em um grafo, onde se pretende separar os nós (as operações) pelos k *data centers* minimizando a soma dos pesos dos arcos que ligam nós que estão em *data centers* diferentes (a soma da quantidade de dados transferidos entre *data centers*).

Ao calcularmos um corte-k em um grafo, pode existir a situação onde "cortamos" vários arcos entre um nó especial, que se encontra num *data center*, e seus filhos, que se encontram noutro *data center*. Diz-se que um nó é especial quando este transfere o mesmo *output* para os seus filhos. Se calcularmos o valor desse corte-k, contabilizamos o custo dessa transferência mais do que uma vez, o que não deveria acontecer, pois esses arcos representam a mesma informação.

Em um trabalho realizado anteriormente, foi criado um algoritmo de corte-k que considera a existência desses nós especiais. Neste trabalho, foram criados dois algoritmos que consideram os nós especiais e a possibilidade de haver nós replicados, caso isto minimize o tráfego de dados entre *data centers*. O primeiro algoritmo considera que existem apenas dois *data centers*, enquanto que o segundo considera a existência de mais do que dois *data centers*. Segundo os resultados experimentais, existem situações em que a replicação permitiu reduzir o tráfego de dados entre *data centers*.

Palavras-chave: grafo, fluxo máximo, corte mínimo, nós especiais, replicação

Abstract

Nowadays, with the increasing amount of data stored in the data centers spread all over the world, it comes the problem of where tasks should be executed in a way that data transfer between data centers is minimized. This problem can be transformed in a minimum k-cut graph problem, where we want to partition the nodes (operations) between the k data centers minimizing the sum of the weights of the edges that connect nodes that are in different data centers (the amount of data that is transferred between data centers).

There is one special case when we are calculating the minimum k-cut, that is when we "cut" the edges between one special node, hosted in one data center, and its children, which are in another data center. We say that a node is special when it transfers the same output to its children. When we calculate the minimum k-cut value, the weight of this transference will be accounted more than once, which should not happen, because the edges represent the same information.

Previously, it was created a k-cut algorithm that takes into consideration the existence of special nodes. In this work, two algorithms were created that consider the special nodes and the possibility of having replicated nodes, if this minimizes the amount of data transferred between data centers. The first algorithm assumes that there are only two data centers, while the second considers the existence of more than two data centers. According to the experimental results, there are some situations where replication has reduced the amount of data transferred between data centers.

Keywords: graph, maximum flow, minimum cut, special nodes, replication

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Capítulo 1

Introdução

1.1 Contexto

1.1.1 Contexto Histórico

A teoria dos grafos foi descoberta em 1736 por Leonard Euler quando resolveu o problema das Sete Pontes em Konigsberg [1]. Esse problema consistia em criar um caminho onde cruzamos cada uma das pontes apenas uma vez.

Euler começou por transformar o mapa da figura 1.1 em um grafo, que pode ser visto na figura 1.2, onde A,B,C,D são os nós do grafo e as pontes são os arcos. Ao passarmos para o grafo, temos que encontrar um circuito que passe por todos os arcos apenas uma vez. Esse caminho terminará no nó onde começarmos. Euler provou que, para isso ser possível, cada nó teria que ter um número par de arcos incidentes e portanto o problema inicial é impossível.

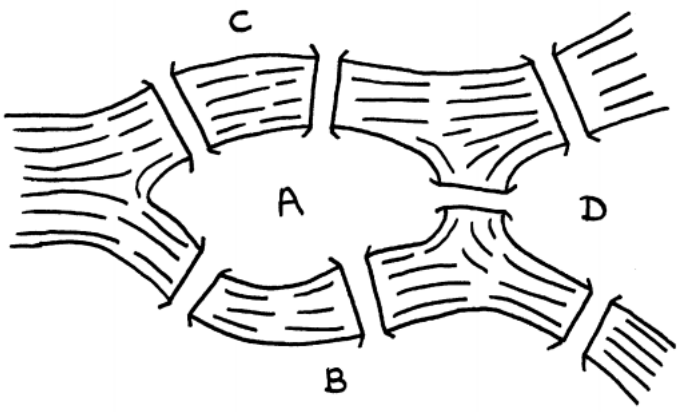


Figura 1.1: Exemplo do problema das sete pontes de Konigsberg (retirado de [1]).

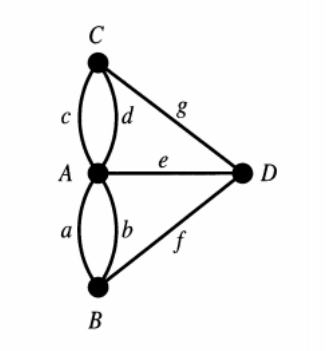


Figura 1.2: Grafo do problema das sete pontes de Konigsberg (retirado de [1]).

No século XX, surgiram dois outros problemas muito conhecidos na teoria dos grafos: o problema do caixeiro viajante [2] e o teorema das quatro cores [3]. O problema do caixeiro viajante consiste em encontrar um caminho em que o caixeiro saia de sua cidade, passe por um grupo de cidades determinadas e volte para casa de forma a que a distância percorrida seja mínima. O teorema das quatro cores diz que quatro cores são suficientes para pintar um mapa (qualquer desenho que seja dividido em regiões) de modo a que regiões vizinhas no mapa sejam pintadas de cores diferentes.

A seguir iremos apresentar alguns exemplos onde aplicamos a teoria dos grafos no nosso quotidiano:

- Encontrar o menor caminho entre duas localidades num mapa;
- *Page Ranking*, algoritmo usado para determinar a importância das páginas *web* em uma pesquisa;
- Representar moléculas;
- Representar os mapas dos transportes de uma cidade, onde as paragens são os nós e os arcos são os caminhos entre as paragens;
- Separar o plano de fundo e o plano da frente de uma imagem.

1.1.2 Contexto do Problema

Nos dias de hoje, analisar grandes quantidades de dados tem-se tornado uma das tarefas mais importantes para as grandes empresas. Pois ao fazer essa análise, estas conseguem, por exemplo, fazer recomendações baseadas no histórico do *browser* e fazer publicidade a um produto que a pessoa possa estar interessada.

Por causa dessas novas abordagens, o volume de informação a ser guardada nas *clouds* tem crescido. O que obrigou as empresas, que prestam esses serviços, a aumentar o número de *data centers*. Por uma questão de melhorar a performance e a disponibilidade, os *data centers* estão localizados em diversas partes do mundo [4]. E por isso, as *frameworks* que têm sido usadas até agora para simplificar o processo de análise de dados não podem

ser usadas, pois essas assumem que toda a informação está em apenas um *data center*, onde a transferência de dados entre nós é uniformemente distribuída, o que não é o que acontece entre *data centers*.

Muitas empresas começaram por tentar resolver esse problema ao centralizar o processo das operações, ou seja, cada *data center* envia os seus dados para um *data center* central e este irá realizar a operação. Para além desse processo centralizado gerar um elevado tráfego de dados entre *data centers*, também viola alguns regulamentos existentes, como por exemplo o da União Europeia que limita a quantidade de dados transferidos entre *data centers* [5].

Por causa disso, as operações passaram a ser processadas nos *data centers* em que os dados se encontram, transferindo apenas os resultados parciais das operações entre os *data centers*. Ao fazermos isso, surge um problema. O problema consiste em conseguir uma forma de separar as operações entre os *data centers* de modo a minimizar a transferência de dados entre estes.

Ao analisarmos este problema, podemos pensar que se trata de encontrar um corte mínimo ou um corte-k mínimo, ou seja, encontrar uma maneira de dividir as operações de forma a que o tráfego de dados seja mínimo entre dois ou $k \geq 3$ *data centers*, respectivamente. Mas existem nós que são especiais: esses nós transferem o mesmo *output* para os seus filhos. Esses nós precisam ser tratados de forma diferente dos restantes; caso contrário, os arcos entre o nó especial e seus filhos seriam contabilizados tantas vezes quantos filhos esse nó tiver, caso façam parte do corte.

Um exemplo deste problema pode ser visto na figura 1.3, onde sabemos em quais *data centers* os dados estão armazenados (DC1 e DC2) e onde queremos obter os resultados (DC3). Precisamos então decidir onde iremos executar as operações necessárias de modo a que a transferência de dados entre *data centers* seja mínima. O problema deste exemplo pode ser transformado no grafo da figura 1.4: onde existem 3 nós (A,B,C) que devem estar localizados cada um em um dos *data centers* (esses nós são os nós terminais) e 2 nós especiais (5 e 10), que são os nós que transferem todo o seu *output* para os seus filhos. Ao determinarmos o corte-k mínimo desse grafo, obtemos o corte da figura 1.5 cuja capacidade é 44, ou seja, a soma dos pesos dos arcos que separam os k conjuntos é $8 + 9 + 13 + 14 = 44$. Se considerarmos que os nós 5 e 10 são nós especiais, ou seja, que os arcos entre esses nós e seus filhos representam a mesma informação, então conseguimos ter um corte-k cuja capacidade é 25, como pode ser visto na figura 1.6.

1.2 Âmbito e Objectivos

Num trabalho realizado anteriormente [6] foi criado um algoritmo de corte-k mínimo para grafos com nós especiais. Neste trabalho, queremos estudar se ter alguns dados replicados em mais do que um *data center* poderá diminuir o tráfego de informação entre *data centers*.

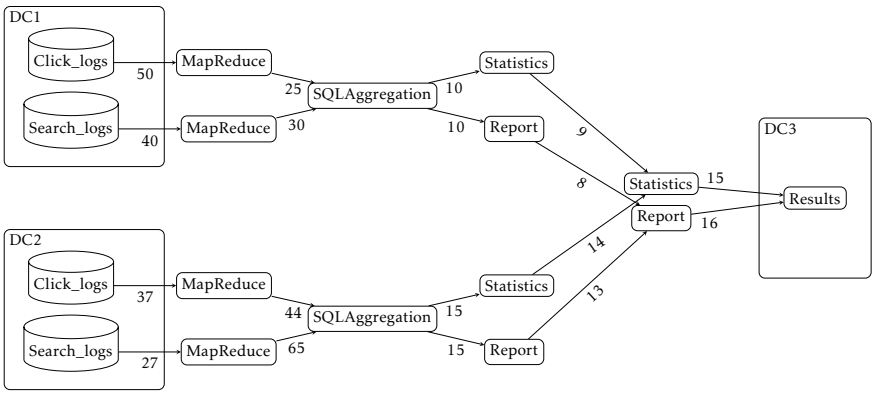


Figura 1.3: Exemplo do problema com nós especiais

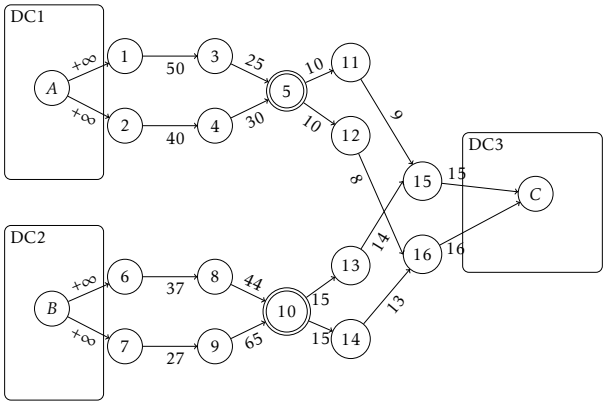


Figura 1.4: Grafo do problema com nós especiais

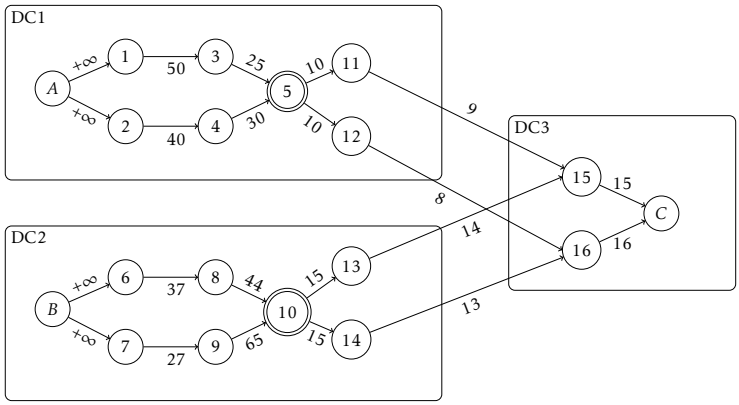


Figura 1.5: Corte-k mínimo sem considerar que os nós 5 e 10 são especiais

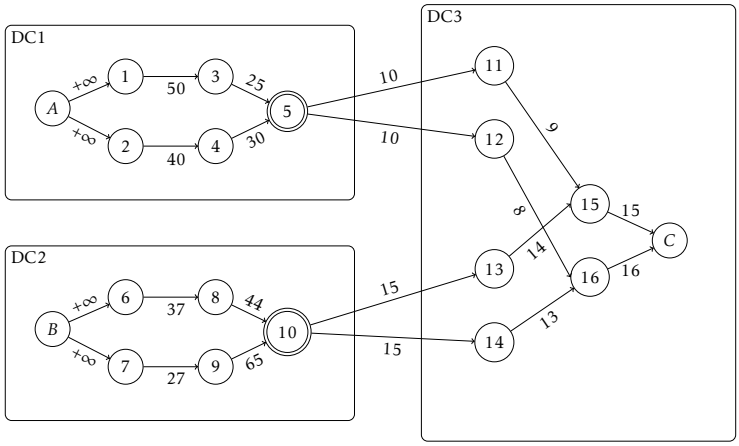


Figura 1.6: Corte-k mínimo a considerar que os nós 5 e 10 são especiais

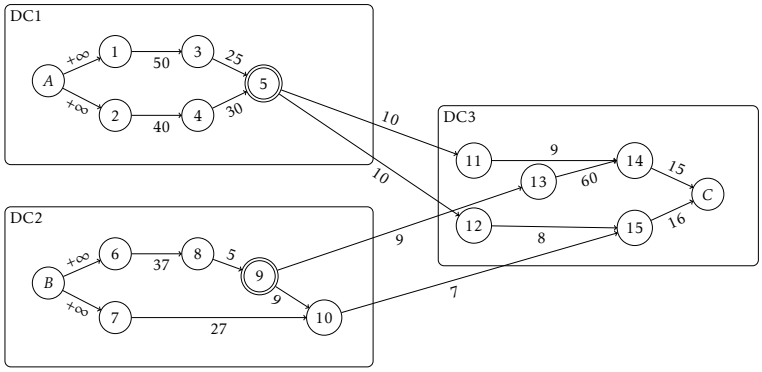


Figura 1.7: Corte-k mínimo com nós especiais (sem replicação)

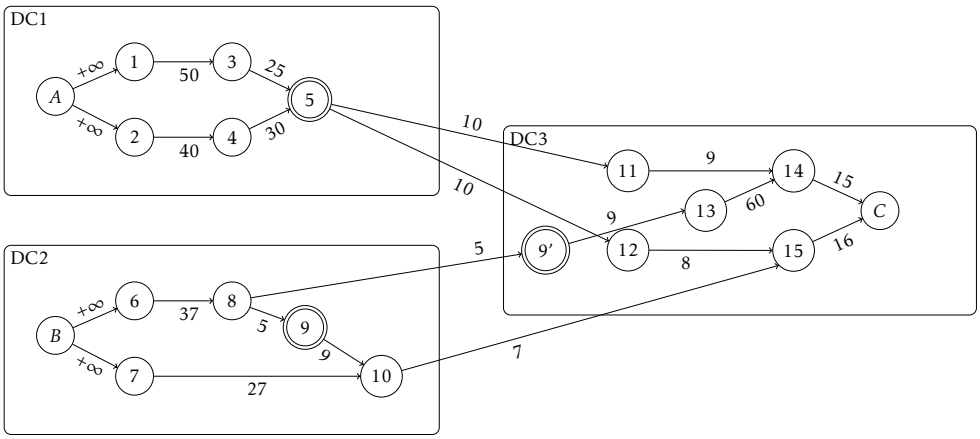


Figura 1.8: Divisão-k com nós especiais (com replicação)

Um exemplo de como a replicação pode diminuir a trasferência de dados entre os *data centers* pode ser visto nas figuras 1.7 e 1.8. Na figura 1.7 está apresentado um corte-k cuja capacidade é 26 onde não existe nenhum nó replicado. Já na figura 1.8, temos uma divisão-k cuja capacidade é 22, pois o nó 9 está replicado em DC2 e DC3. Não podemos dizer que na figura 1.8 temos um corte-k, porque ao determinarmos o corte-k estamos a determinar uma partição dos nós do grafo e em uma partição os conjuntos têm que ser disjuntos entre si, ou seja, um elemento não pode estar em mais do que um conjunto; logo dizemos que criamos uma divisão-k no grafo.

1.3 Principais Contribuições

Neste trabalho tivemos como objetivo a criação de um algoritmo onde ter os dados replicados em mais do que um *data center* minimize o tráfego de dados entre *data centers*. A seguir, apresentamos as principais contribuições:

- Criação de um algoritmo para o problema de divisão mínima com nós especiais;
- Testes experimentais comparando o algoritmo de divisão mínima com o algoritmo de corte mínimo com nós especiais (proposto em [6]);
- Criação de um algoritmo para o problema de divisão-k mínima com nós especiais;
- Testes experimentais comparando o algoritmo de divisão-k mínima com o algoritmo de corte-k mínimo com nós especiais (proposto em [6]).

1.4 Estrutura do Documento

Este documento está dividido em seis capítulos. O capítulo 2 é dedicado ao trabalho relacionado. Começamos por apresentar algumas soluções propostas para o escalonamento de operações em *data centers* na secção 2.1. Na secção 2.2 iremos apresentar os problemas clássicos do fluxo máximo, corte mínimo e corte-k mínimo, mas antes apresentamos algumas definições básicas na secção 2.2.1. Para o problema do fluxo máximo (secção 2.2.2), apresentaremos dois métodos diferentes e alguns dos principais algoritmos que seguem esses métodos. Na secção 2.2.3 definimos corte mínimo, apresentamos um algoritmo clássico e relacionamos fluxos máximos com cortes mínimos. Na secção 2.2.4 será definido corte-k mínimo e apresentado um algoritmo clássico, que é um algoritmo de aproximação. Na secção 2.3 iremos apresentar os problemas de corte mínimo e corte-k mínimo com nós especiais.

O capítulo 3 é dedicado ao algoritmo de divisão mínima com nós especiais. Começamos por apresentar mais detalhadamente o algoritmo de corte mínimo com nós especiais (secção 3.1), pois este algoritmo será essencial para o algoritmo proposto. Na secção 3.2, iremos apresentar o algoritmo proposto e um exemplo da execução deste algoritmo. No

fim deste capítulo, secção 3.3, iremos fazer uma análise à complexidade do algoritmo proposto.

Assim como o capítulo 3, o capítulo 4 começa com uma apresentação mais detalhada do algoritmo de corte-k mínimo com nós especiais (secção 4.1). A secção 4.2 é dedicada à apresentação do algoritmo proposto para o problema de divisão-k mínima com nós especiais. No fim desta secção, apresentamos um exemplo de execução do algoritmo proposto neste capítulo. Na secção 4.3 fazemos uma análise à complexidade do novo algoritmo.

O capítulo 5 é dedicado aos resultados experimentais. Começamos o capítulo com uma secção a apresentar algumas características dos grafos utilizados durante os testes, assim como explicaremos as tabelas que iremos utilizar para apresentar os resultados. A secção 5.2 está dividida em duas partes: a primeira onde iremos apresentar os resultados a comparar o algoritmo de divisão mínima com o algoritmo de corte mínimo com nós especiais [6] em todos os grafos que gerámos; a segunda parte irá comparar os mesmos algoritmos apenas nos grafos onde ocorreu replicação. A secção 5.3 tem a mesma estrutura que a secção anterior, mas ao invés de compararmos os algoritmos de divisão mínima e corte mínimo, iremos comparar os algoritmos de divisão-k mínima e o de corte-k mínimo com nós especiais.

O capítulo 6 é dedicado às conclusões do trabalho realizado e a trabalhos futuros.

Capítulo 2

Trabalho Relacionado

2.1 Escalonamento em Data Centers

Com o aumento dos dados guardados em *data centers*, foram criadas muitas propostas para decidir como escalonar as tarefas nos diversos *data centers*. Separamos algumas dessas propostas em grupos baseado no objetivo que estas propostas pretendem alcançar ao realizar o escalonamento.

Os autores de [7] pretendem minimizar o tempo de execução ao determinarem onde as tarefas devem ser realizadas respeitando a justiça máx-mín (*max-min fairness*). Para isso, o problema foi dividido em subproblemas que são resolvidos por programação linear. Em [8] propõem um algoritmo heurístico que considera a localização, a tranferência e a replicação dos dados de forma a minimizar o tempo de execução.

Em [9] foi proposto uma *framework* que reduz os custos de transferência, armazenamento, computação e latência ao executar as duas fases *MapReduce* em diversos *data centers*. Para conseguir reduzir todos os custos simultaneamente, o problema foi transformado num problema de otimização não linear inteira. TripS [10] é um sistema que determina em que *data center* e em que *tier* as tarefas devem ser realizadas de forma a minimizar os custos de armazenamento. Usam programação linear inteira mista (MILP) para resolver este problema.

Em [11] foi proposto um modelo de processamento de dados de forma a que o sistema seja tolerante a falhas sem que seja adicionado um *overhead* ao replicar os dados. Este modelo diz que os dados devem ser processados em curtos intervalos de tempo. MillWheel [12] é uma *framework* que também pretende que o sistema seja tolerante a falhas. Isso é garantido ao serem feitas verificações de cada chave durante o processamento das mensagens. O que faz com que o sistema seja mais tolerante a falhas mas acrescenta latência nesse processo.

Existem autores que consideram que os recursos no próprio *data center* são mais baratos do que transferir para outro *data center*, por isso têm como principal objetivo reduzir o tráfego de dados entre *data centers*. Em [13] foi proposto uma solução para reduzir o tráfego de dados entre *data centers*, onde os dados estão organizados seguindo o modelo relacional. Usam progamação linear inteira para decidir onde cada tarefa deve ser executada. Para conseguir alcançar esse objetivo, também fazem cache de resultados intermédios, para que no futuro apenas seja preciso transferir a diferença dos resultados, e replicam dados sempre que a replicação melhorar a performance. JetStream [14] é uma solução proposta para reduzir o tráfego de dados entre *data centers* ao fazer a análise de dados em praticamente tempo real, como por exemplo, analisar as imagens de um vídeo. Para isso, os autores propõem que os dados sejam armazenados de forma estruturada para que seja possível agregar os dados que estão relacionados. Também propõem que sejam aplicados filtros para reduzir a quantidade de dados que devem ser transferidos. Nesta solução, os autores não equacionam a possibilidade de replicar dados. Em [5] o problema de reduzir o tráfego de dados entre *data centers* é resolvido através de um particionamento dos nós do grafo. Um particionamento diferente, pois é considerado a existência de nós especiais, nós que transferem todo o seu *output* para seus filhos. Assim como em [14], os autores também não colocam a hipótese de replicar os dados.

2.2 Problemas Clássicos

Nessa secção iremos apresentar os problemas do fluxo máximo, corte mínimo e corte-k mínimo e alguns algoritmos para resolver esses problemas. Antes de apresentarmos os problemas clássicos, precisamos apresentar algumas definições básicas.

2.2.1 Definições Básicas

Um *grafo* é uma estrutura composta por nós e arcos, onde cada arco determina a relação entre dois nós. Seja um grafo $G = (V, E)$, onde V representa o conjunto dos nós e E representa o conjunto de arcos do grafo. Dados os nós v e u, a relação entre eles é descrita pelo par (v, u).

Um grafo diz-se *orientado* [15] se o arco tiver um sentido, ou seja, dados dois nós, v e u, o par (v, u) representa a relação do nó v com o nó u, enquanto que o par (u, v) representa a relação do nó u com o nó v . Enquanto que num grafo *não orientado* [15] o par (u, v) representa a relação do nó u com o nó v e do nó v com o nó u.

Um grafo diz-se *pesado* [15] quando existe um peso ou custo associado ao arco. No contexto deste trabalho, este custo diz-se a *capacidade* do arco, que é representada por $c(u, v)$.

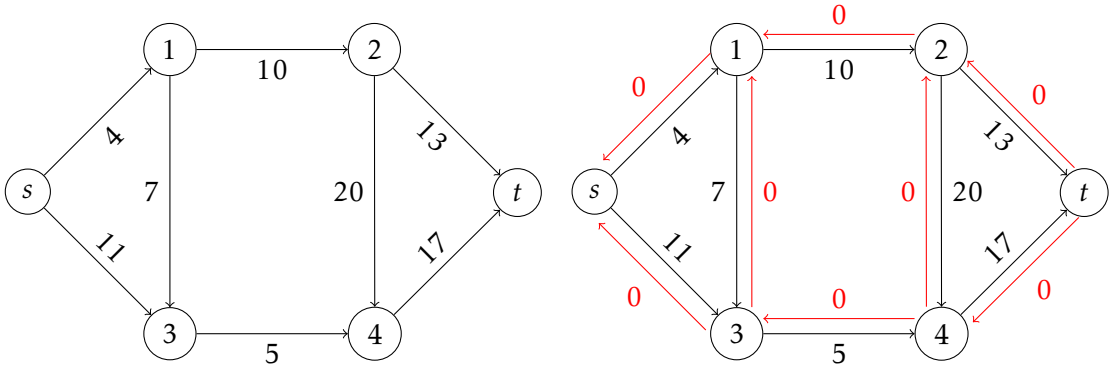
Em um grafo, normalmente, estamos interessados em encontrar um caminho de um nó para outro. Um *caminho* [15] é uma sequência não vazia de nós $n_1, n_2, ..., n_k$ onde, para cada par de nós consecutivos n_{i-1}, n_i, $(n_{i-1}, n_i) \in E$.

2.2.2 Fluxo Máximo

Seja um grafo $G = (V, E)$ orientado e pesado, onde o peso dos arcos é não negativo. Pretende-se calcular um fluxo entre dois nós que têm uma função especial no grafo: a *fonte* s, nó que inicia o percurso do fluxo, e o *dreno* t, o destino do fluxo que foi injetado no grafo. Antes de definir um fluxo, precisamos definir uma rede de fluxos.

A *rede de fluxos* de um grafo $G = (V, E)$ é um novo grafo $M = (V, E')$ que tem todos os arcos de G, $E \subseteq E'$, e onde para quaisquer dois nós, u e v, se o arco $(u, v) \in E$ e $(v, u) \notin E$, então o arco (v, u) é adicionado à rede de fluxos com peso igual a zero.

Na figura 2.1a, podemos observar um grafo original e, na figura 2.1b, a rede de fluxos do grafo original, onde os arcos adicionados estão marcados com a cor vermelha.



(a) Grafo original

(b) Rede de fluxos do grafo original

Figura 2.1: Exemplo de uma rede de fluxos

Fluxo é uma função $f : E \to \mathbb{R}$ que satisfaz as seguintes 3 propriedades:

- *Restrição de Capacidade*: $f(u, v) \leq c(u, v), \forall\ (u, v) \in E$
- *Restrição de Simetria*: $f(u, v) = -f(v, u), \forall\ (u, v) \in E$
- *Conservação do Fluxo*: $\sum_{u \in V} f(u, v) = 0, \forall\ v \in V \setminus \{s, t\}$

O *valor do fluxo* presente no grafo pode ser calculado por: $\sum_{u \in V} f(s, u)$.

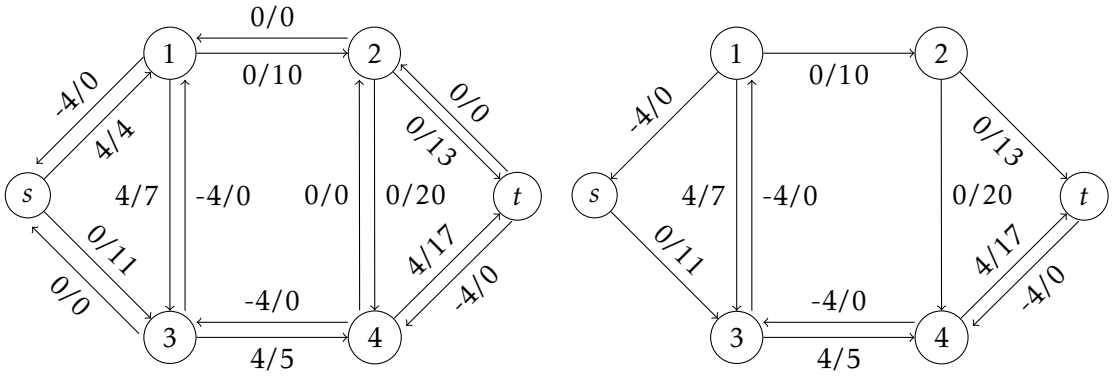
O *problema do fluxo máximo* consiste em encontrar um fluxo da fonte s até ao dreno t cujo valor seja máximo.

Existem diversos métodos e algoritmos para resolver o problema do fluxo máximo. A seguir apresentamos os métodos de Ford-Fulkerson e push-relabel e alguns dos principais algoritmos desses métodos. Antes de apresentar esses métodos, temos que definir o conceito de rede residual.

A *rede residual* de um grafo $G = (V, E)$ induzida por um fluxo f é um novo grafo $N = (V, E'')$, onde para quaisquer dois nós, u e v, se a condição $c(u, v) - f(u, v) > 0$ for verificada, então $(u, v) \in E''$.

Na figura 2.2a, podemos ver um exemplo de uma rede de fluxos, onde foi inserido o fluxo f de 4 unidades no caminho $s \to 1 \to 3 \to 4 \to t$. Nessa figura, podemos observar que

os arcos têm dois números: o da esquerda é o fluxo que passa por esse arco e o da direita é a capacidade desse arco. Por exemplo, o arco que liga o nó 1 ao nó 3 tem 4 unidades de fluxo e 7 de capacidade. Na figura 2.2b, podemos ver a rede residual correspondente (onde "desapareceram" os arcos que "estão cheios").



(a) Rede de fluxos e o fluxo f (b) Rede residual induzida pelo fluxo f

Figura 2.2: Exemplo de uma rede residual

Método de Ford-Fulkerson

O método de Ford-Fulkerson [16] é um dos métodos para resolver o problema do fluxo máximo. Segundo este método, temos que encontrar caminhos entre a fonte e o dreno na rede residual do grafo (onde, inicialmente todos os arcos têm fluxo zero) não sendo especificado como esses caminhos são encontrados.

Quando um caminho p é encontrado, temos que verificar qual a menor capacidade disponível de todos os arcos que pertencem ao caminho: esse será o *resíduo* r desse caminho. Quando sabemos o valor do resíduo de p, temos que atualizar os fluxos de todos os arcos que pertencem a p, ou seja, para todos os arcos $(u,v) \in p$, temos que $f'(u,v) = f(u,v) + r$ e, para os arcos inversos (v,u), temos que $f'(v,u) = f(v,u) - r$, onde f' é o novo fluxo.

Após o fluxo ser atualizado, este processo é repetido enquanto existirem caminhos que possam passar fluxo entre a fonte e o dreno.

Algoritmo de Edmonds-Karp

O algoritmo de Edmonds-Karp [17] segue o método de Ford-Fulkerson. Na listagem 2.1, podemos observar a estrutura do algoritmo de Edmonds-Karp. O algoritmo consiste em encontrar caminhos entre a fonte e o dreno. Esses caminhos são encontrados através de diversas pesquisas em largura, pois cada pesquisa apenas encontra um caminho. Após não serem encontrados mais caminhos com o mesmo comprimento, o algoritmo de pesquisa em largura incrementa o tamanho dos caminhos até não existirem mais caminhos que possam levar fluxo da fonte ao dreno. A pesquisa consiste em explorarmos os nós até encontrarmos o dreno e, se não tivermos mais nós para explorar, o algoritmo acaba sem

```
edmonds_karp(graph, source, sink):
let flow_value be the value of the flow
let flow[u][v]


network <- get_network(graph)

flow_value <- 0
for each edge (n,u) in network.edges()
    flow[n][u] <- 0

while true
    (residue, path) <- breadth_first_search(network, flow, source, sink)
    if residue ⩽ 0
        break
    else
        flow_value <- flow_value + residue
        current_node <- sink
        while current_node ≠ source
            prev_node <- path[current_node]
            flow[prev_node][current_node] <- flow[prev_node][current_node] + residue
            flow[current_node][prev_node] <- flow[current_node][prev_node] - residue
            current_node <- prev_node

return (flow_value, flow)
```

Listagem 2.1: Algoritmo de Edmonds-Karp

sucesso. Explorar um nó significa saber se este já foi visto através de algum outro nó e se é possível enviar fluxo para os nós adjacentes ao nó que está a ser explorado.

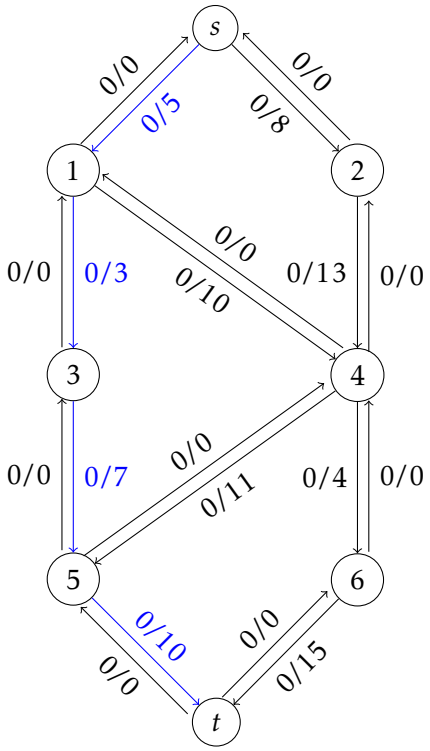
Após cada caminho ser encontrado, temos que atualizar o fluxo do grafo, ou seja, para cada arco que pertence ao caminho encontrado, temos que adicionar o resíduo desse caminho e decrementar esse mesmo valor para os arcos inversos do caminho. Para determinarmos o resíduo de um caminho, calculamos a diferença entre a capacidade de um arco e o fluxo que passa por este, para todos os arcos deste caminho, e escolhemos a diferença que tem menor valor.

Na figura 2.3 é apresentado um exemplo da execução do algoritmo de Edmonds-Karp. Na figura 2.3a, está representada a rede de fluxos do grafo original com o primeiro caminho encontrado, assinalado a azul. A figura 2.3b é a rede de fluxos após ser inserido o fluxo no primeiro caminho encontrado. Na figura 2.3c está representada a rede de fluxos após serem encontrados os outros caminhos, que são: $s \to 1 \to 4 \to 5 \to t$, $s \to 2 \to 4 \to 5 \to t$ e $s \to 2 \to 4 \to 6 \to t$. Um fluxo máximo nesse grafo tem o valor 13.

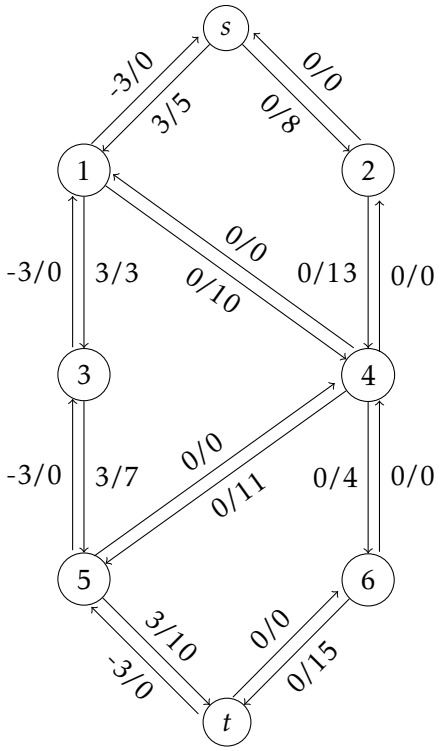
O algoritmo de Edmonds-Karp tem complexidade temporal de $O(|V|\,|E|^2)$ [15], onde $|V|$ é o número de nós e $|E|$ é o número de arcos no grafo.

Algoritmos Precursores do Método Push-Relabel

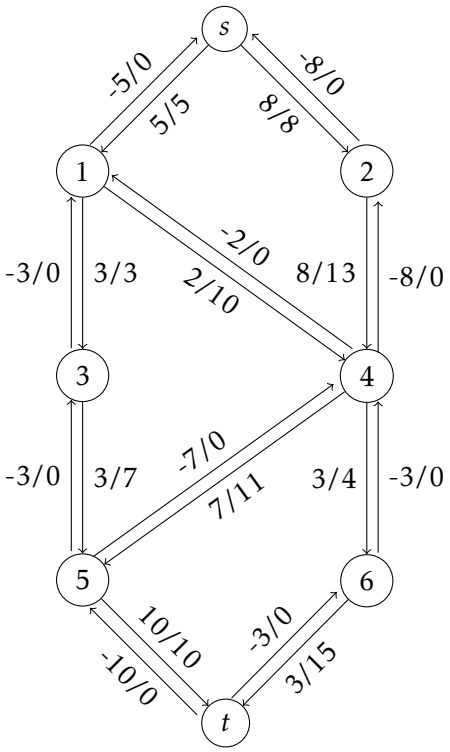
O algoritmo de Dinitz [18] é muito semelhante ao algoritmo de Edmonds-Karp, a única diferença é que, ao invés de ser procurado apenas um caminho por ciclo, são procurados todos os caminhos do mesmo tamanho, criando assim *layers*, ou seja, cada *layer* representa o tamanho dos caminhos encontrados. Logo, se um nó $n \in L_3$, significa que o nó n pertence



(a) Rede de fluxos inicial com o primeiro caminho encontrado assinalado a azul



(b) Rede de fluxos após processar o primeiro caminho



(c) Rede de fluxos final após processar todos os caminhos encontrados

Figura 2.3: Exemplo do algoritmo de Edmonds-Karp

a um caminho de tamanho 3. Este algoritmo possui complexidade temporal de $O(|V|^2\,|E|)$ [18].

Outro algoritmo importante é o algoritmo de Karzanov [19]. Karzanov foi quem introduziu o conceito de pré-fluxo. Um *pré-fluxo* "é um fluxo" onde a quantidade de fluxo que chega a um nó pode ser maior ou igual à quantidade de fluxo que sai desse nó, violando assim a propriedade de conservação de fluxo. O algoritmo consiste em inserir um pré-fluxo enquanto existirem caminhos do mesmo tamanho. Quando não existirem mais caminhos desse determinado tamanho, o tamanho do caminho é aumentado e o processo é repetido até não ser possível aumentar o tamanho dos caminhos. Então o excesso de pré-fluxo, ou seja, a diferença entre o fluxo que entra no nó e o que sai, é mandado de volta para a fonte. E assim o pré-fluxo se torna fluxo e é máximo. Este algoritmo possui complexidade temporal de $O(|V|^3)$ [19].

Cherkasky [19] também foi um dos responsáveis por criar um algoritmo para resolver o problema do fluxo máximo. O seu algoritmo consiste na união do algoritmo de Dinitz e do algoritmo de Karzanov, ou seja, as *layers* foram juntadas em blocos e em cada um desses blocos de *layers* foi usado o algoritmo de Dinitz. Ao fazer essa união, conseguiu que a complexidade temporal passasse a ser $O(|V|^2\sqrt{|E|})$ [19].

Os algoritmos de Dinitz, Karzanov e Cherkasky levaram à criação do método *push-relabel*.

Método Push-Relabel

Diferente do método de Ford-Fulkerson, o método push-relabel usa o conceito de pré-fluxo que foi introduzido pelo algoritmo de Karzanov. No método push-relabel, cada nó tem duas características importantes, o excesso e a altura. O *excesso* é a diferença entre a quantidade de pré-fluxo que chega a um nó e a quantidade de pré-fluxo que sai desse mesmo nó. A *altura* do nó é usada para determinar para quais nós o pré-fluxo pode ser enviado; cada nó só pode fazer *push* se a sua altura for maior do que a altura do nó para o qual pretende enviar o fluxo. O método começa com a altura da fonte igual ao número de nós presentes no grafo e todos os outros nós, inclusive o dreno, com altura igual a zero.

Neste método, existem duas operações: *push* e *relabel*. A operação de *push* consiste em enviar o excesso de um nó para um de seus nós adjacentes, respeitando a capacidade do arco entre eles. A operação de *relabel* consiste em atualizar as alturas dos nós, ou seja, fazer com que um nó tenha a altura necessária para fazer *push*. Logo, se um nó u, cuja altura é h_u, quer fazer *push* para o nó v e a altura de v é h_v, onde $h_v \geq h_u$, então $h_u = h_v + 1$.

Algoritmos Push-Relabel

O método push-relabel pode ser implementado de diversas formas, fazendo com que as complexidades temporais variem. De seguida, iremos apresentar algumas dessas implementações e suas complexidades temporais.

```
push_relabel(graph, source, sink):
let excess[n] be the excess of a vertex
let height[n] be the height of a vertex
let queue be an empty queue of the vertices to be explored
let flow [n][u]

network <- get_network(graph)

for each node n in network.vertices()
    excess[n] <- 0
    height[n] <- 0

height[source] <- network.vertices().size()

for each edge (n,u) in network.edges()
    flow[n][u] <- 0

for each node n in network.out_adjacent_nodes(source)
    flow[source][n] <- network.capacity(source,n)
    flow[n][source] <- - network.capacity(source,n)
    excess[n] <- network.capacity(source,n)
    if (n ≠ sink)
        queue.enqueue(n)

while not queue.empty()
    head <- queue.peek()
    aux <- +∞
    for each n in network.out_adjacent_nodes(head)
        if network.capacity(head, n) - flow[head][n] > 0
            if height[head] > height[n]
                push(network, flow, excess, head, n)
                if not is_in_queue(n) and n ≠ source and n ≠ sink
                    queue.enqueue(n)
                if excess[head] = 0
                    queue.dequeue(head)
                    break
            else
                aux <- min(aux, height[n])

    if excess[head] > 0
        height[head] <- aux + 1

flow_value <- 0
for each node n in network.in_adjacent_nodes(sink)
    flow_value <- flow_value + flow[n][sink]
return (flow_value, flow)
```

Listagem 2.2: Algoritmo push-relabel

```
push(network, flow , excess, n1, n2):
push_value <- min(excess[n1], network.capacity(n1, n2) - flow[n1][n2])
flow[n1][n2] <- flow[n1][n2] + push_value
flow[n2][n1] <- - flow[n1][n2]
excess[n1] <- excess[n1] - push_value
excess[n2] <- excess[n2] + push_value
```

Listagem 2.3: Operação de push

Ao implementarmos o algoritmo push-relabel com uma fila FIFO (*first in first out*), onde nessa fila serão guardados os nós que deverão ser explorados, temos que a complexidade temporal é de $O(|V|^3)$ [19], onde $|V|$ é o número de nós no grafo. Nesse caso, o algoritmo teria a estrutura que pode ser observada na Listagem 2.2. Começamos por inicializar as alturas dos nós: a fonte com o número de nós no grafo e a de todos os outros com zero. Os excessos dos nós são inicializados a zero. Depois, inserimos um fluxo igual à capacidade de cada arco em todos os arcos que partem da fonte, atualizamos o valor do excesso desses nós e inserimos estes na fila, caso não sejam o dreno.

A seguir, temos o ciclo principal, onde continuamos enquanto a fila não estiver vazia. Neste ciclo, vemos o nó que está no início da fila e percorremos todos os nós adjacentes a esse nó. Ao percorrer os nós adjacentes, verificamos se podemos enviar pré-fluxo naquele arco; caso seja possível, verificamos se a altura do nó no início da fila é maior do que a do nó adjacente. Caso seja, fazemos *push*, cujo código pode ser visto em Listagem 2.3, e adicionamos o nó para o qual fizemos *push* na fila. Caso contrário, verificamos qual o valor da altura que o nó do início da fila tem que ter para poder fazer *push*. Quando percorremos todos os nós adjacentes ao primeiro nó da fila e não conseguimos fazer *push* para mais nenhum nó e o primeiro nó da fila ainda tem excesso, então fazemos *relabel* da altura do primeiro nó da fila, ou seja, atualizamos a altura do nó para que este possa fazer *push* para algum dos seus nós adjacentes. Quando sairmos do ciclo, calculamos o valor do fluxo obtido ao executar o algoritmo.

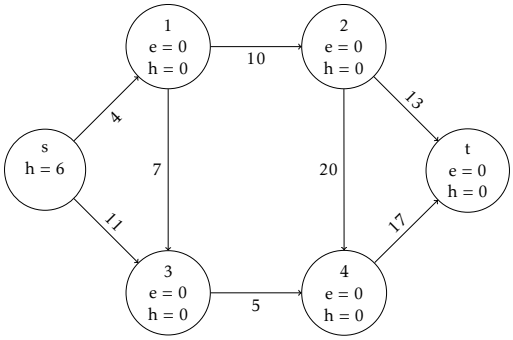
Na figura 2.4, podemos observar uma execução do algoritmo push-relabel. Na figura 2.4b, podemos ver a rede residual após a fonte fazer *push* para os nós adjacentes. Nas figuras 2.4c a 2.4g podemos ver a rede residual após os nós 1, 3, 2, 4 terem feito *relabel* e *push*. E na figura 2.4g podemos ver o grafo final. O valor do fluxo máximo encontrado pelo algoritmo push-relabel é 9.

Em comparação com o algoritmo apresentado anteriormente, cuja complexidade temporal é de $O(|V|^3)$ [19], se usarmos uma *dynamic tree*, passamos a ter $O\left(|V||E| \log \frac{|V|^2}{|E|}\right)$ [19]. É possível conseguir essa complexidade, pois é utilizada uma nova função de *push*, *tree-push*. Essa função faz *push* no caminho todo da árvore, mas esta apenas pode ser executada quando a árvore que tem os dois nós envolvidos no *push* tem no máximo $\frac{|V|^2}{|E|}$ nós. Se escolhermos o nó com maior altura para fazer *push* do seu excesso, temos que a complexidade temporal é $O(|V|^2\sqrt{|E|})$ [20].

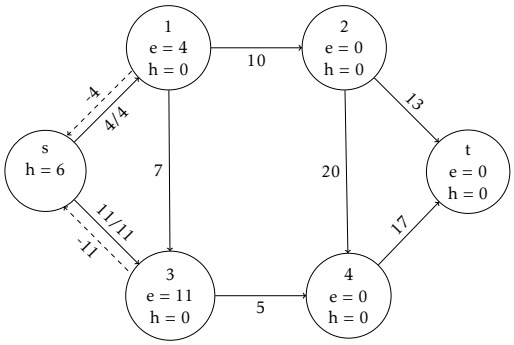
Outros Algoritmos

Existem muitos outros algoritmos que resolvem o problema do fluxo máximo e que não seguem o método de Ford-Fulkerson nem o de *push-relabel*. Iremos agora apresentar alguns desses algoritmos.

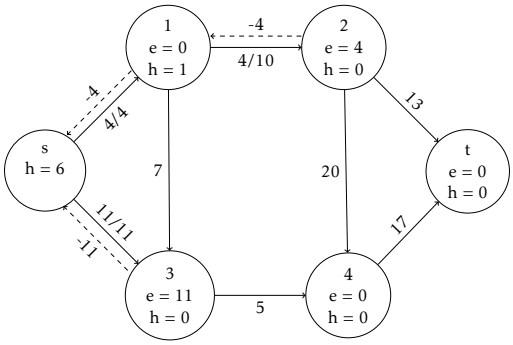
Um dos algoritmos é o algoritmo de James B. Orlin [21] cuja complexidade é $O(|V||E|)$, onde $|V|$ é o número de nós e $|E|$ é o número de arcos. Essa complexidade é possível, pois o fluxo máximo é calculado em um grafo compacto, ou seja, um grafo onde alguns nós e



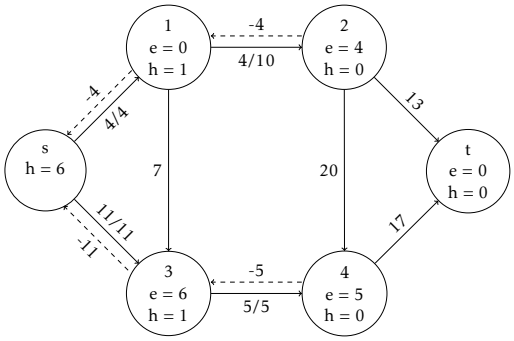
(a) Grafo inicial



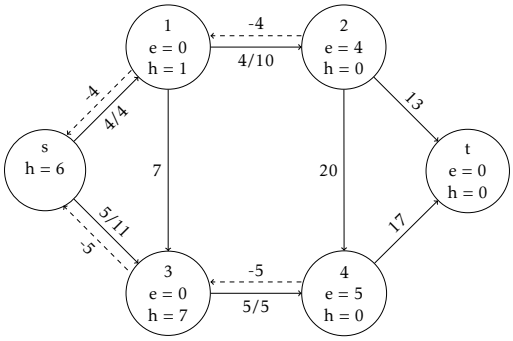
(b) Grafo após a fonte fazer push para os nós adjacentes



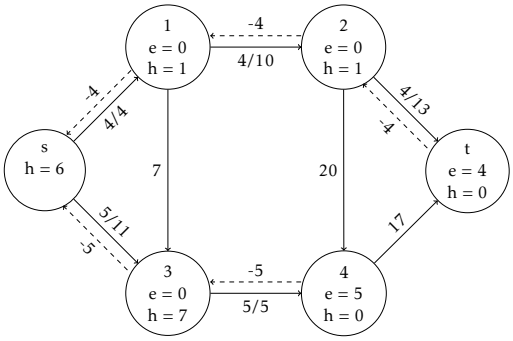
(c) Grafo após o nó 1 fazer relabel e push para o nó 2



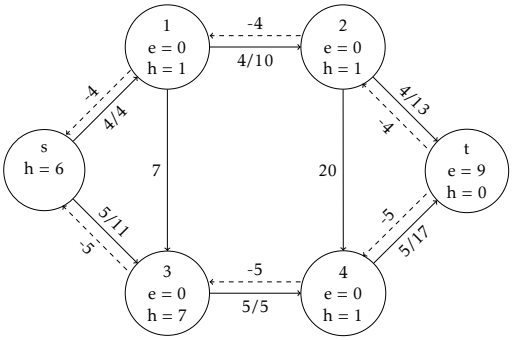
(d) Grafo após o nó 3 fazer relabel e push para o nó 4



(e) Grafo após o nó 3 fazer relabel e push para a fonte



(f) Grafo após o nó 2 fazer relabel e push para o dreno



(g) Grafo após o nó 4 fazer relabel e push para o dreno

Figura 2.4: Exemplo do algoritmo push-relabel

arcos são unidos. Essa complexidade passa a ser $O\left(\frac{|V|^2}{\log|V|}\right)$ no caso de $|E| = O(|V|)$.

Outro algoritmo é o algoritmo de V.M. Malhotra, M. Pramodh Kumar e S.N. Mareshwari que tem complexidade de $O(|V|^3)$ [22]. Esse algoritmo consiste em escolher um nó a cada iteração. Será este nó que irá determinar o caminho entre a fonte e o dreno. Este nó tem um valor associado, que representa o máximo de fluxo que o nó pode receber. Este fluxo é encaminhado para a fonte e o dreno.

Um outro algoritmo importante é o algoritmo de Andrew V. Goldberg e Satash Rao, que tem complexidade $O\left(\min(|V|^{\frac{2}{3}}, \sqrt{|E|}) * |E| * \log\left(\frac{|V|^2}{|E|}\right)\log U\right)$ [23], onde U é a maior capacidade dos arcos. As capacidades têm que ser inteiros. Este algoritmo é dividido em fases, onde nestas fases são calculados cortes s-t.

2.2.3 Corte Mínimo

Sejam um grafo $G = (V, E)$ não orientado e pesado, com pesos não negativos, e dois nós s e t. Pretende-se encontrar uma partição $\{S, T\}$ de V, onde $s \in S$ e $t \in T$. Essa partição nos indica qual o conjunto de arcos que devem ser removidos para termos os dois conjuntos de nós, S e T, completamente "desligados" [15]. Esse conjunto, chamado o *conjunto de corte*, pode ser descrito por:

$$cut\_set(S, T) = \{\ (u, v) \in E \mid u \in S, v \in T\ \}.$$

A *capacidade do corte* é a soma do peso dos arcos que fazem parte do conjunto de corte:

$$c(S, T) = \sum_{(u,v) \in cut\_set(S,T)} c(u, v).$$

O *problema do corte mínimo* consiste em encontrar um corte cuja capacidade seja mínima.

Existe uma relação entre fluxos máximos e cortes mínimos. Para estes se relacionarem, é preciso passar o grafo não orientado para um grafo orientado.

Seja um grafo $G = (V, E)$ não orientado e pesado. O grafo orientado e pesado correspondente a G é o grafo $M = (V, E')$ onde, para todos os pares de nós $(u, v) \in E$, então $(u, v) \in E'$ e $(v, u) \in E'$ com capacidade igual à capacidade de (u, v) em G.

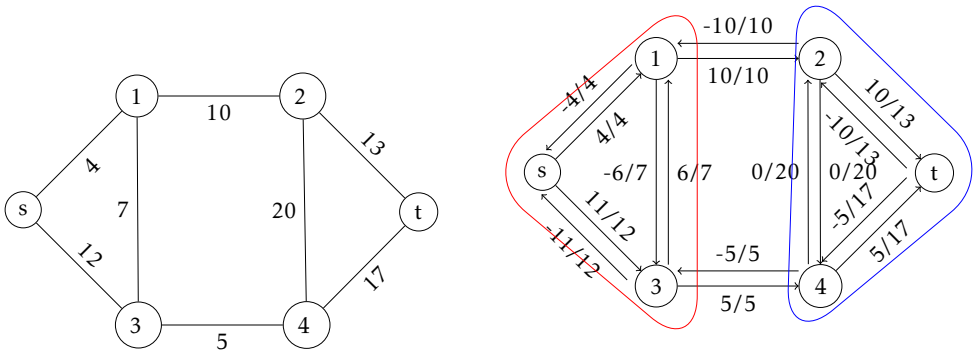
Para simplificar, vamos considerar que todos os grafos não orientados são orientados, havendo dois arcos com sentidos opostos por cada arco do grafo original. No entanto, as figuras poderão apresentar a versão não orientada, que é mais compacta.

Teorema do Fluxo-Máximo Corte-Mínimo : o valor dos fluxos máximos é igual à capacidade dos cortes mínimos [15].

Para determinar um corte mínimo de um grafo $G = (V, E)$, seguimos o algoritmo cujo pseudo-código é apresentado na Listagem 2.4. Primeiramente, através da função max_flow, é calculado um fluxo máximo do grafo G. Após calcularmos um fluxo máximo de G, temos que fazer uma pesquisa na rede residual induzida pelo fluxo máximo e assim calculamos um corte de G. Calculamos o corte da seguinte forma: se existir um caminho

```
min_cut(graph, source, sink):
let partition[n] be the set of a vertex

(flow_value, flow) <- max_flow(graph, source, sink)

seen <- search(network, flow, source)

for each node n in graph.vertices()
    if !seen[n]
        partition[n] <- sink
    else
        partition[n] <- source

return (partition, flow_value)
```

Listagem 2.4: Algoritmo de corte mínimo



(a) Grafo original

(b) Corte mínimo no grafo original

Figura 2.5: Exemplo de um corte mínimo

entre a fonte e um determinado nó, então esse nó pertence ao conjunto da fonte; caso contrário, então o nó pertence ao conjunto do dreno.

Na figura 2.5a podemos observar um grafo original. Na figura 2.5b, podemos ver o grafo orientado correspondente ao grafo da figura 2.5a, onde está exemplificado um corte mínimo. Este corte tem capacidade 15 e os conjuntos $S = \{s, 1, 3\}$ e $T = \{2, 4, t\}$ definem a partição $\{S, T\}$.

2.2.4 Corte-k Mínimo

Sejam um grafo $G = (V, E)$ não orientado e um conjunto $L = \{l_1, l_2, ..., l_k\} \subseteq V$ com $k \geq 3$ nós, chamados *nós terminais*. É importante ressaltar que um corte-k apenas está definido para $k \geq 3$, pois para $k = 2$ se trata de um corte. Pretende-se encontrar uma partição $\{L_1, L_2, ..., L_k\}$ de V, onde $l_1 \in L_1$, $l_2 \in L_2$,..., $l_k \in L_k$. Assim como no corte mínimo, essa partição nos indica qual o conjunto de arcos que devem ser removidos para termos os conjuntos $L_1, L_2, ..., L_k$ "desligados" entre si [24]. Esse conjunto é descrito por:

$$cut\_k\_set(L_1, L_2, ..., L_k) = \{ (u, v) \in E \mid u \in L_i,\ v \in L_j,\ i \neq j \}.$$

A *capacidade do corte-k* é a soma do peso dos arcos que fazem parte do conjunto $cut\_k\_set(L_1, L_2, ..., L_k)$:

$$c(L_1, L_2, ..., L_k) = \sum_{(u,v) \in cut\_k\_set(L_1, L_2, ..., L_k)} c(u, v).$$

Assim como no problema do corte mínimo, no *problema do corte-k mínimo* também se pretende encontrar um corte-k cuja capacidade seja mínima.

Na figura 2.6, podemos ver um exemplo de um corte-k mínimo, onde $k = 3$ e $L = \{A, B, C\}$. A capacidade desse corte-k é de 28.

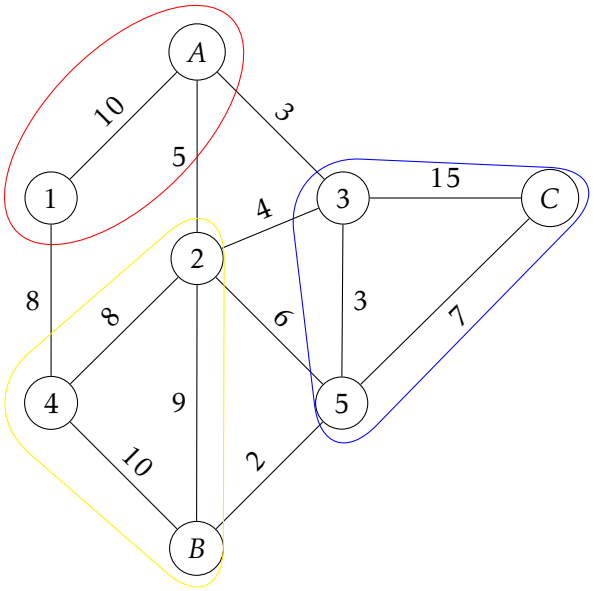


Figura 2.6: Exemplo de um corte-k mínimo

Para determinarmos um corte-k do grafo $G = (V, E)$, quando $k \geq 3$, iremos calcular um corte mínimo para cada um dos nós terminais e a "união" desses cortes irá ser o corte-k. Para calcularmos o corte mínimo, temos que fazer uma cópia do grafo original, para cada um dos nós terminais, e adicionar um nó auxiliar à cópia do grafo original, que será chamado *sink*. Ao nó auxiliar, *sink*, chegarão arcos que partem de todos os nós terminais, excepto do nó terminal que estamos a considerar como *source*, com capacidade $+\infty$. Depois de calcularmos o corte mínimo, iremos atualizar o vector com o corte-k, ou seja, para todos os nós que pertencem ao conjunto da *source* no corte mínimo, dizemos que estes pertencem ao conjunto do nó terminal que temos como *source* no corte-k. Também verificamos se esse corte tem a maior capacidade de todos os cortes mínimos. E repetimos esse processo para todos os nós terminais. Após calcularmos todos os cortes mínimos, adicionamos ao conjunto do corte com maior capacidade os nós que não pertencem a nenhum conjunto. Por fim, calculamos a capacidade do corte-k. Para calcular a capacidade do corte-k, percorremos todos os arcos que pertencem ao grafo original e verificamos se os nós que esse arco liga pertencem ao mesmo conjunto. Se não pertencerem, adicionamos

```
min_k_cut(graph, terminals):
let partition[n] be the partition of the vertices
let sink be a new vertex in the copies of the graph
let max_cut_capacity be the value of the highest min cut
let max_terminal be the terminal with the highest min cut
let cut_capacity be the capacity of a cut

max_cut_capacity <- 0


for each node n in graph.vertices()
    partition[n] <- ∅

for each node source in terminals
    graph_copy <- clone(graph)
    graph_copy.add_node(sink)
    for each node r in terminals
        if r ≠ source
            graph_copy.add_edge(r, sink, +∞)

    (cut, cut_capacity) <- min_cut(graph_copy, source, sink)

    for each node n in graph.vertices()
        if cut[n] = source
            partition[n] <- source

    if cut_capacity > max_cut_capacity
        max_cut_capacity <- cut_capacity
        max_terminal <- source

for each node n in graph.vertices()
    if partition[n] = ∅
        partition[n] <- max_terminal

cut_capacity <- 0
for each edge (u,v) in graph.edges()
    if partition[u] ≠ partition[v]
        cut_capacity <- cut_capacity + graph.capacity(u,v)

return (partition, cut_capacity)
```

Listagem 2.5: Algoritmo de corte-k mínimo

a capacidade desse arco à capacidade do corte-k. A estrutura do algoritmo de corte-k mínimo descrito anteriormente pode ser vista na Listagem 2.5.

O algoritmo apresentado é um algoritmo de aproximação, devido ao facto de o corte-k mínimo ser um problema NP-difícil quando $k \geq 3$. Logo não é garantido que iremos encontrar um corte-k mínimo, apenas que iremos ter um corte-k cuja capacidade é próxima da de um corte-k mínimo. Por isso, temos que definir o rácio de aproximação.

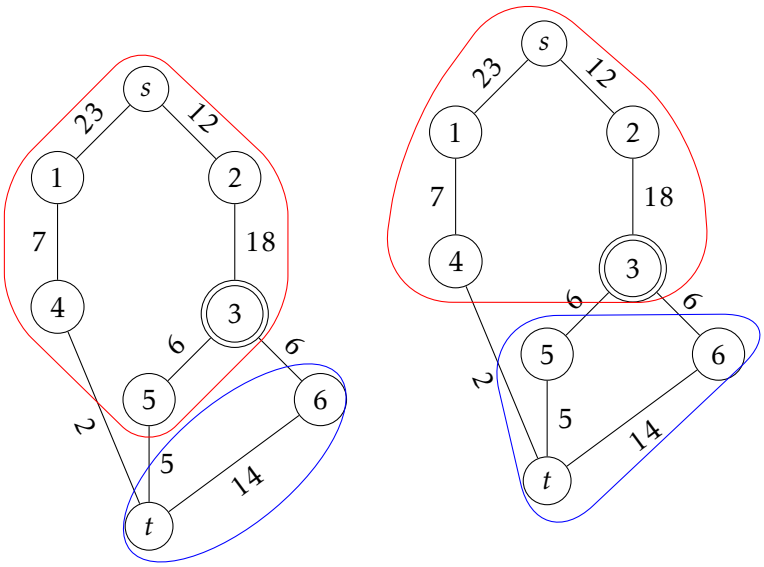
O rácio de aproximação é o quociente entre a capacidade da solução encontrada, S, e a capacidade da solução ótima, S^*. Essa relação não pode ser maior que um determinado α, $\frac{S}{S^*} \leq \alpha$. No nosso caso, temos que $\alpha = 2 - \frac{2}{k}$ [25], onde k é a quantidade de nós terminais.

2.3 Problemas com Nós Especiais

Nesta secção, iremos apresentar os problemas de corte mínimo e corte-k mínimo com nós especiais. Esses nós são especiais pois transferem o mesmo *output* para os seus filhos. Iremos representar esses nós por σ. Os filhos dos nós especiais são os nós que irão receber como *input* os dados transferidos pelos nós especiais. Iremos representar os filhos dos nós especiais por $child(\sigma)$.

2.3.1 Corte Mínimo

Quando tentamos encontrar um corte mínimo, usando o algoritmo clássico, em um grafo com esses nós especiais, o peso dos arcos entre um nó especial σ e seus n filhos será contabilizado tantas vezes quantos filhos esse nó tiver $(n \times c(\sigma, y))$, caso façam parte do corte. Isto pode ser observado na figura 2.7, onde o nó 3 é o nó especial e 5 e 6 são seus filhos. Se não considerarmos que o nó 3 é um nó especial, temos que a capacidade dos cortes mínimos é 13, como na figura 2.7a. Mas se considerarmos que os arcos entre o nó 3 e seus filhos representam a mesma informação, devido ao *Dataflow Forking*, então apenas contabilizamos o peso 6 uma vez no corte da figura 2.7b, que tem capacidade 8 e é o corte mínimo pretendido.



(a) Corte mínimo sem considerar que 3 é um nó especial

(b) Corte mínimo considerando que o nó 3 é especial

Figura 2.7: Exemplo de como um nó especial pode influenciar a capacidade do corte

O sistema Pixida [5] tenta resolver este problema através da criação de nós extra entre os nós especiais e seus filhos para que o peso do arco que liga o nó especial σ aos seus filhos seja contabilizado no máximo uma vez no corte.

A formalização deste novo problema passa por considerar um conjunto de grafos, chamados instâncias do grafo original. Uma *instância* $G = (V, E)$ do grafo original $G^{or} = (V^{or}, E^{or})$, com o conjunto de nós especiais $\Sigma \subseteq V^{or}$, é obtida através dos seguintes passos:

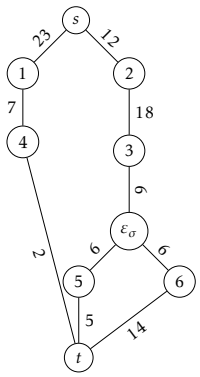


Figura 2.8: Exemplo da instância inicial G^{init} do grafo da figura 2.7

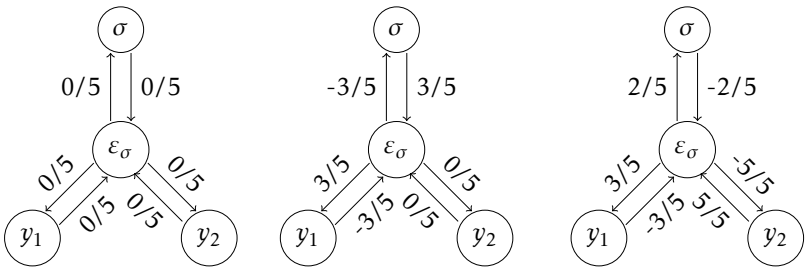
- Todos os nós presentes no grafo original também pertencem à instância.
- Todos os arcos presentes no grafo original que não ligam um dos nós especiais a um de seus filhos também pertencem à instância.
- Para cada nó especial, pode ser adicionado um ou dois nós extras:
  - Se todos os filhos de σ ficarem no mesmo conjunto, então é adicionado apenas um nó extra ε_σ; este é ligado ao nó especial σ e a todos os filhos de σ.
  - Se os filhos de σ ficarem em conjuntos diferentes, $\{Y_1, Y_2\}$, então são adicionados dois nós extras ε_σ^1 e ε_σ^2; o nó extra ε_σ^1 está ligado a σ e a cada filho em Y_1, enquanto que o nó extra ε_σ^2 está ligado a σ e a cada filho em Y_2.

Nos dois casos, o peso dos arcos adicionados é igual ao peso do arco que liga σ aos seus filhos no grafo original.

O *problema do corte mínimo com nós especiais* consiste em encontrar a instância do grafo original G^{or} em que o corte será mínimo, tendo em consideração os cortes das outras instâncias. Para isso, o algoritmo proposto é separado em 4 partes: criação da instância inicial, obtenção do fluxo base, criação da instância canónica e cálculo do corte mínimo.

Primeiramente, devemos criar a instância inicial G^{init} do grafo original G^{or}. Na *instância inicial* G^{init} é adicionado um nó extra ε_σ entre cada nó especial σ e seus filhos, ou seja, para o grafo da figura 2.7, teriamos o grafo da figura 2.8.

Agora que já temos a instância inicial, G^{init}, temos que calcular o seu fluxo base. O algoritmo usado para calcular o fluxo base é semelhante ao algoritmo de *Edmonds-Karp*, apresentado na listagem 2.1. A única diferença é no tipo de caminhos que são procurados. No algoritmo proposto, estamos interessados em encontrar caminhos restritos, ou seja, caminhos que ligam a fonte ao dreno e satisfazem a condição de manter sempre a mesma direção do fluxo entre o nó extra ε_σ e os filhos do nó especial, $child(\sigma)$. Por exemplo, dados dois nós, y_1 e y_2 que são filhos de σ, se o fluxo entre o nó extra e y_1 é positivo, $f(\varepsilon_\sigma, y_1) > 0$, então o fluxo entre o nó extra e y_2 também é positivo ou nulo, $f(\varepsilon_\sigma, y_2) \geqslant 0$. Na figura



(a) Grafo sem fluxo (b) Fluxo válido (c) Fluxo inválido

Figura 2.9: Exemplo de fluxos válidos e não válidos

2.9, podemos ver exemplos de fluxos válidos e inválidos. Na figura 2.9b podemos ver um exemplo de fluxo válido, obtido ao injetar 3 unidades de fluxo no caminho $\sigma \to \varepsilon_\sigma \to y_1$, na rede residual da figura 2.9a. Esse fluxo é válido porque mantém a direção do fluxo entre o nó extra e os filhos de σ. O que não acontece na figura 2.9c, pois temos fluxo a descer pelo arco $(\varepsilon_\sigma, y_1)$ e temos fluxo a subir pelo arco $(y_2, \varepsilon_\sigma)$. Podemos reparar que o fluxo representado na figura 2.9c poderia ser obtido ao injetar 5 unidades de fluxo no caminho $y_2 \to \varepsilon_\sigma \to \sigma$, na rede de fluxos da figura 2.9b. Por isso, $y_2 \to \varepsilon_\sigma \to \sigma$ não é um caminho restrito.

Para podermos entender o conceito dos caminhos restritos, primeiro temos que definir os conjuntos Σ_0, Σ_{down} e Σ_{up}. Seja f um fluxo na instância inicial G^{init}, σ um nó especial e ε_σ o nó extra correspondente.

- O nó especial σ pertence ao conjunto $\Sigma_0 \subseteq \Sigma$ se não existir fluxo em nenhum dos arcos entre o nó extra ε_σ e os filhos do nó especial σ:

$$\forall u \in child(\sigma)\, f(\varepsilon_\sigma, u) = 0$$

- O nó especial σ pertence ao conjunto $\Sigma_{down} \subseteq \Sigma$ se apenas existir fluxo a descer entre o nó extra ε_σ e os filhos do nó especial σ:

$$\exists u \in child(\sigma)\, f(\varepsilon_\sigma, u) > 0 \; e \; \forall u \in child(\sigma)\, f(\varepsilon_\sigma, u) \geqslant 0$$

- O nó especial σ pertence ao conjunto $\Sigma_{up} \subseteq \Sigma$ se apenas existir fluxo a subir entre o nó extra ε_σ e os filhos do nó especial σ:

$$\exists u \in child(\sigma)\, f(u, \varepsilon_\sigma) > 0 \; e \; \forall u \in child(\sigma)\, f(u, \varepsilon_\sigma) \geqslant 0$$

É importante notar que estes conjuntos Σ_0, Σ_{down} *e* Σ_{up} são disjuntos dois a dois, pois o fluxo entre o nó especial σ e o seu nó extra ε_σ satisfaz as seguintes propriedades que são incompatíveis: se $\sigma \in \Sigma_0$, então $f(\sigma, \varepsilon_\sigma) = 0$; se $\sigma \in \Sigma_{down}$, então $f(\sigma, \varepsilon_\sigma) > 0$; e se $\sigma \in \Sigma_{up}$, então $f(\sigma, \varepsilon_\sigma) < 0$.

Seja f um fluxo na intância inicial $G^{init} = (V^{init}, E^{init})$ *e* $u \in V^{init}$. Um *caminho restrito* p a partir da fonte s até ao nó u na rede residual é um caminho simples de s até u que respeita as seguintes restrições para cada nó extra ε_σ que pertence à p [26]:

- **Down Section**: se $\sigma \to \varepsilon_\sigma \to y$ pertence ao caminho p e $y \in child(\sigma)$, então $\sigma \in \Sigma_{down} \cup \Sigma_0$ ou $f(y, \varepsilon_\sigma) > 0$.

  No segundo caso, se $f(y, \varepsilon_\sigma) < f(\varepsilon_\sigma, \sigma)$, então a capacidade disponível do arco $(\varepsilon_\sigma, y) = f(y, \varepsilon_\sigma)$.

- **Up Section**: se $y \to \varepsilon_\sigma \to \sigma$ pertence ao caminho p e $y \in child(\sigma)$, então $\sigma \in \Sigma_{up} \cup \Sigma_0$ ou $f(\varepsilon_\sigma, y) > 0$.

  No segundo caso, se $f(\sigma, \varepsilon_\sigma) > f(\varepsilon_\sigma, y)$, então a capacidade disponível do arco $(y, \varepsilon_\sigma) = f(\varepsilon_\sigma, y)$.

- **Up-Down Section**: se $y_1 \to \varepsilon_\sigma \to y_2$ pertence ao caminho p e $y_1, y_2 \in child(\sigma)$, então $f(\varepsilon_\sigma, y_1) > 0$ ou $f(y_2, \varepsilon_\sigma) > 0$.

  No primeiro caso, a capacidade disponível do arco $(y_1, \varepsilon_\sigma) = f(\varepsilon_\sigma, y_1)$.

  No segundo caso, a capacidade disponível do arco $(\varepsilon_\sigma, y_2) = f(y_2, \varepsilon_\sigma)$.

Ao calcularmos o fluxo base, temos a informação de quais filhos devem estar juntos do nó especial σ e quais devem estar separados. Tendo essa informação em consideração, é criada a *instância canónica* G^{can} do grafo original G^{or}. Nesse passo, é analisado, para cada nó especial σ, se deve ser adicionado um segundo nó extra ou não. Esta decisão é baseada no facto de existirem caminhos restritos para os filhos dos nós especiais. Caso existam caminhos para todos ou para nenhum dos filhos do nó especial, então só é preciso um nó extra ε_σ; caso contrário, são precisos dois nós extras, ε_σ^1 e ε_σ^2.

Para finalizar, precisamos determinar o corte mínimo. Para isso, executamos o algoritmo de *Edmonds-Karp* na instância canónica G^{can}, mas ao invés de começarmos com um fluxo de valor zero, começamos com o fluxo base determinado anteriomente.

Se considerarmos o corte mínimo encontrado na instância canónica, (S, T), e o corte mínimo para o problema com nós especiais, (S^*, T^*), então temos que

$$c(S, T) \leqslant c(S^*, T^*) + \sum_{\sigma \in \Sigma} c(\sigma)$$

onde $c(\sigma)$ é o peso dos arcos entre o nó especial σ e os seus filhos.

Existe um problema no algoritmo apresentado anteriormente. Esse problema pode ser visto no exemplo da figura 2.10 retirado da tese de mestrado [6].

Nesse exemplo, podemos ver um grafo onde já foram encontrados os caminhos $s \to y2 \to t$ e $s \to 1 \to y1 \to t$. Ao executar a pesquisa em largura com as mudanças necessárias para encontrar apenas caminhos restritos, não seriam encontrados mais caminhos entre a fonte e o dreno. Se observarmos o grafo, podemos perceber que ainda existem dois caminhos que respeitam as restrições necessárias. Esses caminhos são:

$$s \to y2 \to extra1 \to sp1 \to 1 \to 2 \to sp2 \to extra2 \to y3 \to t$$
$$s \to y2 \to extra2 \to sp2 \to 2 \to 1 \to sp1 \to extra1 \to y1 \to t.$$

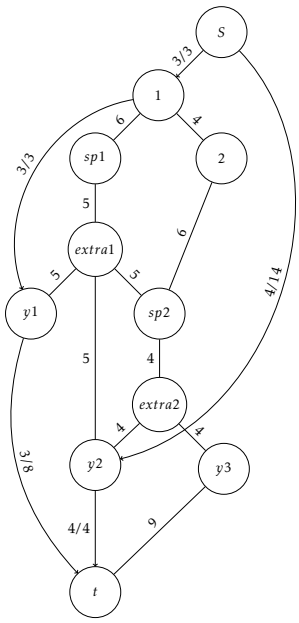


Figura 2.10: Exemplo do problema do algoritmo Pixida (retirado de [6]).

Esses caminhos não são encontrados, pois ao chegar nos nós 1 e 2 não são encontrados mais nós por serem explorados. Este problema acontece devido ao facto de os nós que são vistos quando tentamos subir por nós extras cujos nós especiais são do tipo Σ_0 serem marcados da mesma forma que nos restantes casos.

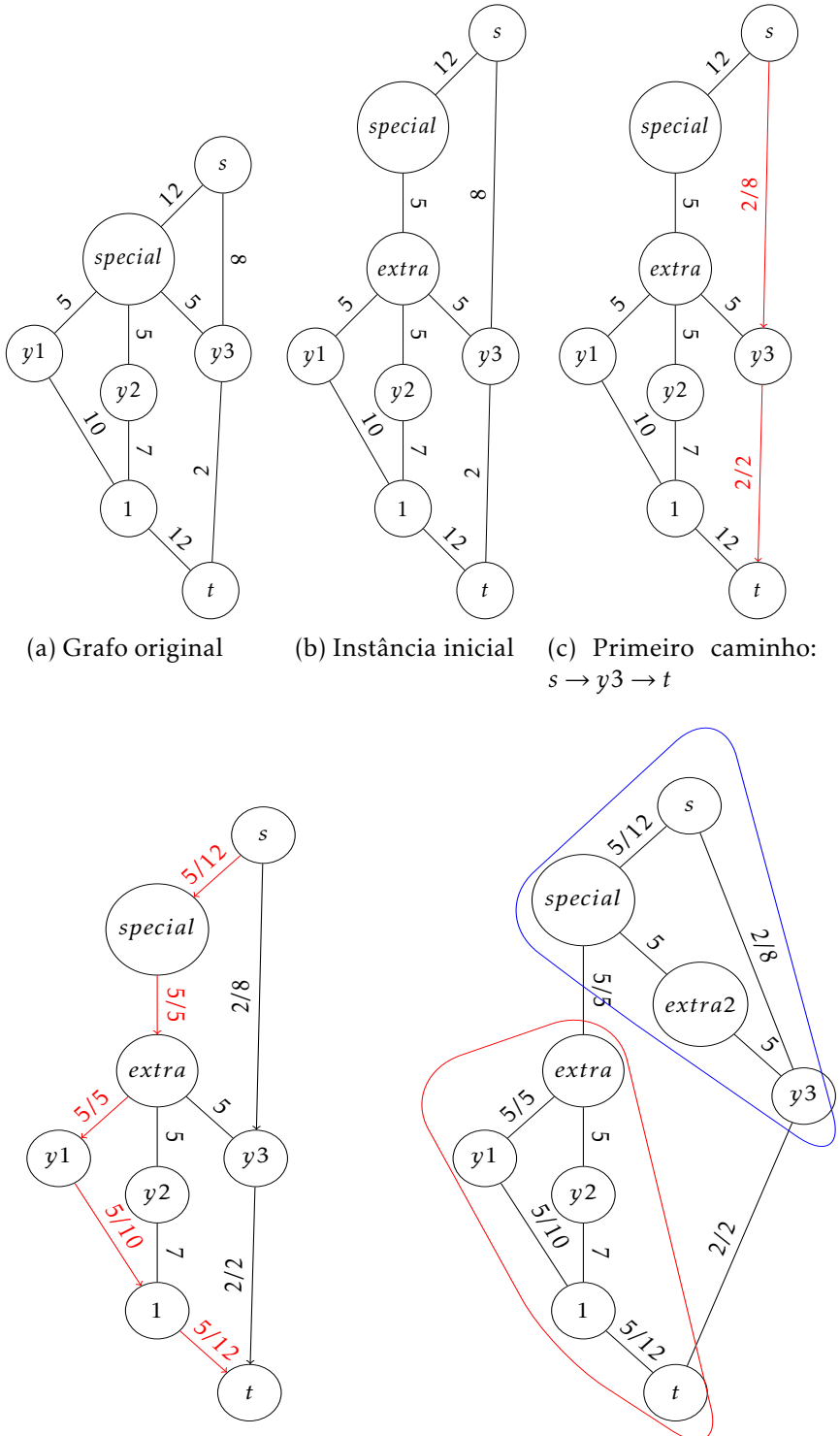
Então, temos que marcar os nós que são vistos por caminhos que sobem por um nó especial em Σ_0 com uma marca que indica de que nó especial estes vieram. Ao colocarmos essas marcas, temos no pior caso $2^{|\Sigma|}$ marcas diferentes, o que faz com que o algoritmo tenha complexidade exponencial.

Por isso, em [6] foi proposto um novo algoritmo, onde cada nó é visitado no máximo duas vezes: no estado *Up*, quando foi encontrado a subir por um nó especial em Σ_0, ou no estado "normal". Esse algoritmo segue a regra de que os caminhos que sobem por um nó especial em Σ_0 têm menos prioridade do que os outros caminhos.

Portanto, é preciso fazer algumas alterações na pesquisa para encontrar um caminho restrito. A primeira mudança está no cálculo do resíduo do caminho. A capacidade disponível de um arco tem que ser o mínimo entre a diferença da capacidade do arco com o fluxo que já passa nesse arco e o valor do fluxo que esse arco pode receber de modo a não mudar a direção do fluxo nos nós especiais. A segunda mudança é termos que marcar um nó com o estado *Up* quando o caminho sobe por um nó especial em Σ_0. Essa marcação nos indica se devemos adicionar o nó no início ou fim da fila. Com isso, garantimos que os nós com estado "normal" são explorados antes dos nós com o estado *Up*.

Este algoritmo tem complexidade temporal $O(f_b|E| + |V||E|^2)$ [6], onde f_b é o valor do fluxo base. Não se sabe se este algoritmo é um algoritmo de aproximação.

Na figura 2.11 pode ser visto um exemplo da execução do algoritmo apresentado, retirado da tese de mestrado [6]. Na figura 2.11a temos o grafo original com um nó



(a) Grafo original

(b) Instância inicial

(c) Primeiro caminho: $s \rightarrow y3 \rightarrow t$

(d) Segundo caminho: $s \rightarrow special \rightarrow extra \rightarrow y1 \rightarrow 1 \rightarrow t$

(e) Instância canónica e corte final

Figura 2.11: Exemplo do algoritmo de corte mínimo com nós especiais (retirado de [6]).

especial. A instância inicial (figura 2.11b) é obtida pela introdução de um nó extra entre o nó especial e seus filhos.

A seguir, temos que calcular o fluxo base. O fluxo base será calculado através de pesquisas em largura, onde são encontrados os caminhos $s \rightarrow y3 \rightarrow t$ e $s \rightarrow special \rightarrow extra \rightarrow y1 \rightarrow 1 \rightarrow t$, como pode ser visto nas figuras 2.11c e 2.11d, respectivamente. Depois do fluxo base, precisamos construir a instância canónica. Para isso, temos que encontrar todos os nós a que conseguimos chegar a partir da fonte por caminhos restritos. Esses são: $\{s, special, y3\}$. Como apenas conseguimos chegar a um filho do nó especial, então temos que adicionar um segundo nó extra, entre o nó especial e $y3$, como pode ser visto na figura 2.11e.

Por último, apenas temos que calcular o corte mínimo na instância canónica, onde obtemos a seguinte partição: $\{\{s, special, extra2, y3\}, \{t, extra, y1, y2, 1\}\}$, cuja capacidade do corte é 7.

De acordo com os resultados experimentais apresentados em [6], o sistema Pixida e o algoritmo proposto em [6] (que marca os nós encontrados durante a pesquisa com os estados Up e "normal") calcularam um corte ótimo em 99,6% dos casos onde os grafos são pequenos (ou seja, grafos que têm no máximo 60 nós). Em 87,6% dos casos, esses dois algoritmos e o algoritmo de Edmonds-Karp (algoritmo da secção 2.2.2) encontraram cortes mínimos (equivalentes); em 12% dos casos, só os algoritmos concebidos para nós especiais retornaram uma solução ótima. Ao compararmos os tempos de execução, o algoritmo de [6] é 3,7 vezes mais lento que o algoritmo clássico e o sistema Pixida é 4,4 vezes mais lento. Para grafos grandes (grafos com 1000 nós, onde a solução ótima não é conhecida), os três algoritmos encontraram cortes com igual capacidade em 83,6% dos casos, enquanto que os dois algoritmos para nós especiais calcularam cortes equivalentes entre si mas com capacidade inferior à do corte computado pelo algoritmo de Edmonds-Karp em 16,4% dos casos. Os tempos de execução do sistema Pixida são aproximadamente o dobro dos tempos do algoritmo clássico, enquanto que o algoritmo proposto em [6] é 1,5 vezes mais lento.

2.3.2 Corte-k Mínimo

O problema do corte-k mínimo com nós especiais consiste em encontrar um corte-k cuja capacidade seja mínima, considerando que a capacidade dos arcos entre um nó especial σ e seus filhos é contabilizada uma só vez quando σ pertence ao conjunto de um nó terminal l e algum dos seus filhos pertence ao conjunto de outro nó terminal l', independentemente do número de filhos de σ que pertencem a l'.

Para resolver este problema, foi proposto um algoritmo semelhante ao apresentado na secção 2.2.4. O algoritmo proposto [6] e o da secção 2.2.4 possuem algumas diferenças. Ao invés de usarmos o algoritmo de corte mínimo, usamos o algoritmo de corte mínimo com nós especiais e não ignoramos o corte que tem a maior capacidade; ao invés disso, todos os cortes são usados e posteriormente são resolvidas possíveis incompatibilidades.

Existem dois tipos de incompatibilidades que temos que resolver: nós que nenhum nó terminal quer e nós que mais do que um nó terminal quer. Para fazer isso, primeiramente temos que separar os nós com o mesmo problema em grupos (que se assemelham a componentes conexas). Para um nó u fazer parte de um grupo X, é preciso que alguma das duas situações seja verdade:

- Existe um arco entre o nó u e algum nó que já pertence ao grupo

$$\exists v \in X, (u, v) \in E;$$

- O nó u tem um irmão que pertence ao grupo X

$$\exists \sigma \in \Sigma,\ \exists v \in X, \{u, v\} \subseteq child(\sigma).$$

Iremos começar por resolver o problema dos nós que nenhum nó terminal quer, onde temos que agrupar os nós adjacentes e identificar quais os terminais que fazem fronteira com cada um desses nós desse grupo.

Após resolver o problema dos nós que nenhum nó terminal quer, temos que resolver o problema dos nós que mais do que um nó terminal quer. Para isso, começamos por encontrar os grupos. Encontrar grupos consiste em encontrar nós que tenham conflito com pelo menos dois nós terminais e que tenham algum nó terminal em comum entre eles. Caso exista mais do que um grupo, temos que juntar os grupos adjacentes. Dois grupos são adjacentes se existir um arco entre eles, ou se existirem filhos de um nó especial nesses dois grupos.

Por fim, calculamos a capacidade do corte caso o grupo pertença ao conjunto de cada um dos nós terminais que são adjacentes. O conjunto do nó terminal que produzir o menor valor de corte será aquele a que o grupo irá pertencer.

Esse algoritmo tem complexidade temporal $O(k f_b |E| + |V|^4)$ [6], onde f_b é o maior valor de um fluxo base, $|V|$ é o número de nós e $|E|$ é o número de arcos no grafo.

Nas figuras 2.12 e 2.13, podemos observar um exemplo de execução do algoritmo de corte-k mínimo com nós especiais, retirado da tese de mestrado [6]. Nesse exemplo temos os nós terminais $\{A, B, C, D\}$ e os nós especiais $\{sp1, sp2\}$ cujos filhos são $\{y1, y2, y3\}$ e $\{y4, y5\}$, respectivamente. Nas figuras 2.12a a 2.12d são apresentados os cortes de cada nó terminal. Ao unirmos os cortes dos nós terminais temos algumas incompatibilidades, que podem ser vistas na figura 2.12e. Os nós $\{1, sp1\}$ pertencem aos conjuntos dos nós terminais C e D, o nó $sp2$ pertence aos conjuntos dos nós A e B e os nós $\{2, 4, y1\}$ não pertencem a nenhum conjunto.

Depois de calcularmos os cortes de cada nó terminal, temos que resolver os conflitos existentes. Começamos por tratar dos nós que não pertencem a nenhum conjunto. No nosso caso, apenas temos um grupo de nós com esse problema ($\{2, 4, y1\}$). Para tratar desses nós temos que encontrar os nós terminais que fazem fronteira com o grupo; neste caso, são os nós terminais $\{B, C, D\}$, como pode ser visto na figura 2.12f.

A seguir, temos que procurar pelos grupos de nós que apresentam algum conflito. Neste exemplo, encontramos dois grupos: $\{1, 2, 4, sp1, y1\}$ em conflito com os nós terminais $\{B, C, D\}$ e $\{sp2\}$ em conflito com os nós terminais $\{A, B\}$, como pode ser visto na figura 2.13a.

Como temos mais do que um grupo em conflito, temos que tentar juntar os grupos adjacentes. No nosso caso, os dois grupos são adjacentes e têm o nó terminal B em comum. Então unimos esses dois grupos em um grupo. Os nós $\{1, 2, 4, sp1, y1, sp2\}$ pertencem ao novo grupo que está em conflito com todos os nós terminais, como é apresentado na figura 2.13b.

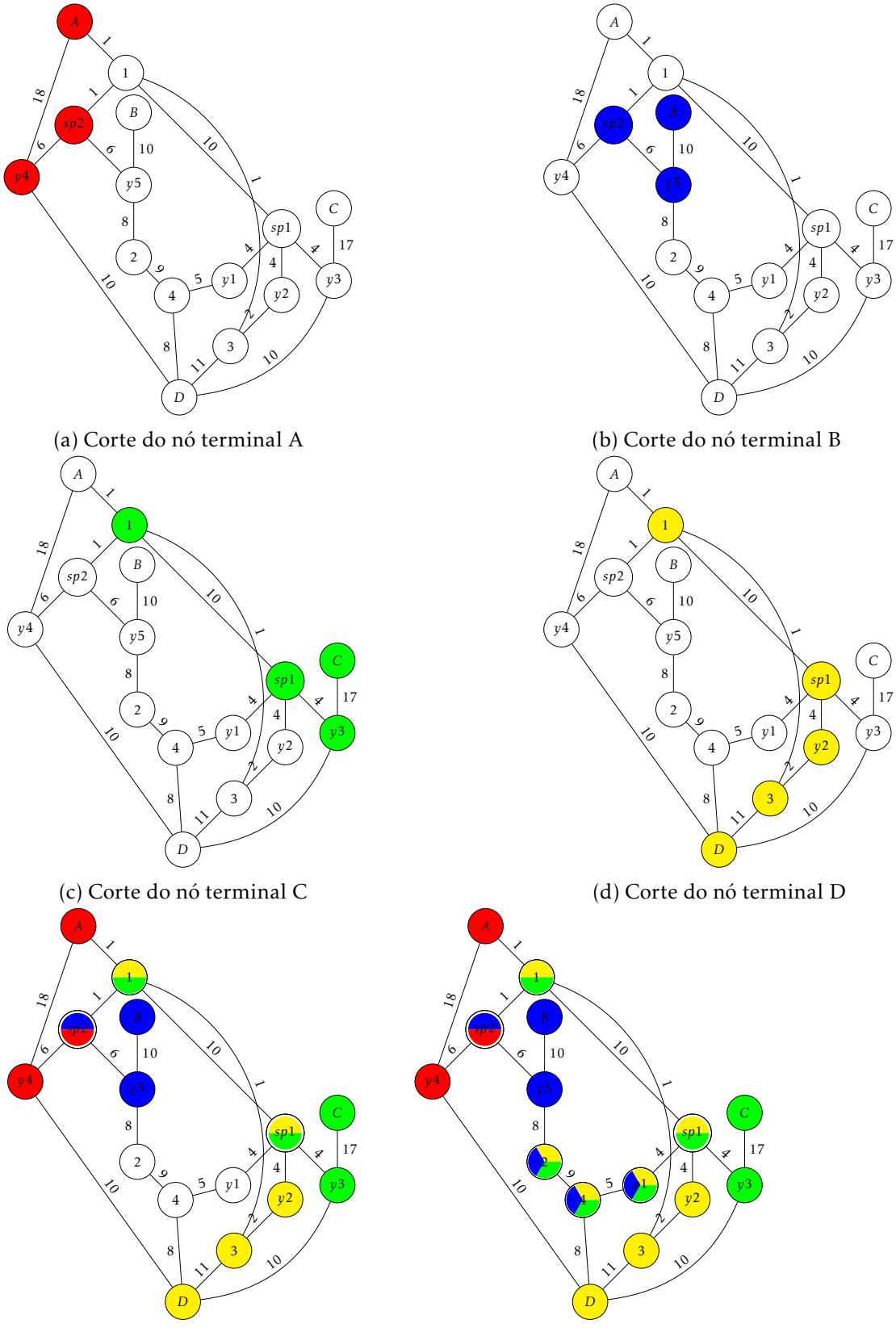
Por último, verificamos a capacidade do corte-k caso o grupo em conflito pertença a cada um dos conjuntos dos nós terminais que façam fronteira com esse grupo. Ao calcularmos os valores, verificamos que a redução na capacidade do corte-k obtida caso o grupo pertença ao conjunto de A é 7, de B é 14, de C é 4 e de D é 13. Como o maior valor encontrado é quando o grupo pertence ao conjunto do nó terminal B, esse é o escolhido. O corte final tem capacidade 44 e pode ser visto na figura 2.13c.

Os resultados experimentais apresentados em [6] comparam 4 algoritmos:

- (A1) o algoritmo clássico (algoritmo apresentado na Listagem 2.5), usando o algoritmo de Edmonds-Karp;
- (A2) uma variante do algoritmo clássico, que, em vez de atribuir os nós que nenhum nó terminal quer ao conjunto do nó terminal cujo corte mínimo tem a maior capacidade, atribui esses nós a um nó terminal que minimiza a capacidade do corte-k retornado;
- o algoritmo proposto em [6] (descrito na secção 2.3.2) com duas alternativas: os cortes mínimos (com nós especiais) são calculados com (A3) o sistema Pixida ou com (A4) o algoritmo proposto em [6].

Para os 250.000 grafos com até 60 nós, os quatro algoritmos calcularam cortes-k com igual capacidade em 55% dos casos; em 35,9% dos casos, os dois algoritmos concebidos para vértices especiais retornam cortes-k equivalentes entre si e melhores que os calculados por (A1) e (A2); em 8,8% dos casos (A2), (A3) e (A4) retornaram cortes-k com capacidade inferior à dos cortes-k retornados por (A1). Ao compararmos o tempo de execução, (A4) é 3,1 vezes mais lento do que (A1), enquanto que (A3) é 3,6 vezes mais lento.

Para os 20.000 grafos grandes (grafos com 1000 nós), os 4 algoritmos encontram cortes com igual capacidade em 19% dos casos, enquanto que (A3) e (A4) encontram cortes cuja capacidade é inferior à dos cortes computados por (A1) e (A2) em 80,9% dos casos. O tempo de execução de (A3) é 2,8 vezes mais lento do que (A1), enquanto que (A4) é 2,2 vezes mais lento.



(a) Corte do nó terminal A

(b) Corte do nó terminal B

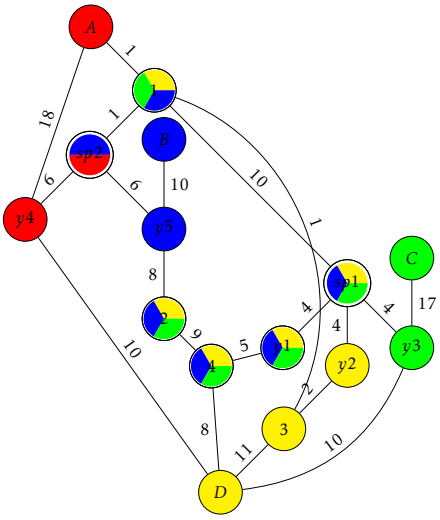
(c) Corte do nó terminal C

(d) Corte do nó terminal D

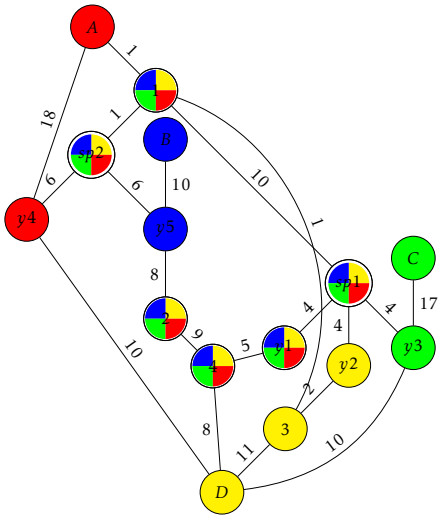
(e) União dos cortes

(f) Grafo após tratar dos nós que nenhum terminal quer

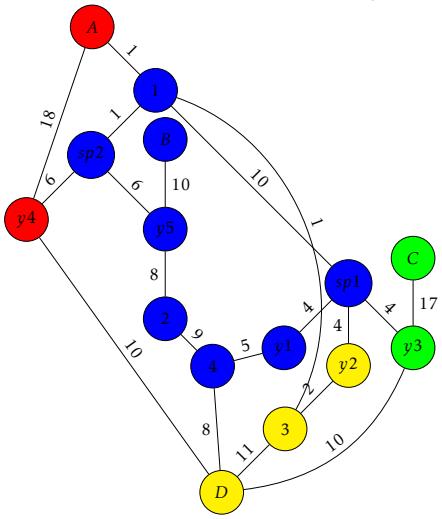
Figura 2.12: Exemplo do algoritmo de corte-k mínimo com nós especiais (retirado de [6])



(a) Grafo após encontrar os grupos $\{1, 2, 4, sp1, y1\}$ e $\{sp2\}$



(b) Grafo após juntar os grupos $\{1, 2, 4, sp1, y1\}$ e $\{sp2\}$



(c) Corte final

Figura 2.13: Continuação do exemplo do algoritmo de corte-k mínimo com nós especiais (retirado de [6]).

Capítulo 3

Divisão Mínima com Nós Especiais

3.1 Algoritmo de Corte Mínimo com Nós Especiais

Nesta secção iremos explicar mais detalhadamente alguns dos processos realizados no algoritmo de corte mínimo com nós especiais [6] que serão fundamentais para o algoritmo proposto.

Como podemos ver na listagem 3.1, o algoritmo começa com a criação da instância inicial. Esta instância é criada ao adicionarmos um nó extra entre cada nó especial e seus filhos. Para isso, temos que remover os arcos entre o nó especial e seus filhos e adicionar os arcos entre o nó extra e os filhos do nó especial. Assim como temos que adicionar um arco entre o nó especial e o nó extra. O processo da criação da instância inicial pode ser visto na listagem 3.2.

```
special_min_cut(graph, source, sink):
init_inst <- build_initial_instance(graph)
base_flow <- compute_base_flow(init_inst, source, sink)
access_base <- get_accesible_nodes(init_inst, base_flow, source)
can_inst <- build_canonical_instance(init_inst, access_base)

return min_cut(can_inst, source, sink)
```

Listagem 3.1: Algoritmo de corte mínimo com nós especiais (retirado de [6])

A seguir iremos calcular o fluxo base, que pode ser visto na listagem 3.3. O fluxo base é calculado por uma função semelhante ao algoritmo de Edmonds-Karp. As principais diferenças são: os caminhos encontrados são restritos; a função que encontra os caminhos retorna um vector com o par do nó anterior e o tipo deste nó ao invés de só o nó anterior.

A seguir ao fluxo base, iremos verificar quais os nós que são acessíveis a partir da fonte. Dizemos que um nó acessível a partir da fonte é um nó *querido* pela fonte. O processo para

```
build_initial_instance(graph):
init_inst <- build_network(graph)

for each special in graph.get_special_nodes()

    let extra be a new extra vertex to be added to the graph

    init_inst.add_extra_node(extra, special)

    children <- graph.get_children(special)
    let child be any element in children
    cap <- graph.capacity(special, child)

    for each child in children
        init_inst.remove_edge(child, special)
        init_inst.remove_edge(special, child)
        init_inst.add_edge(extra, child, cap)
        init_inst.add_edge(child, extra, cap)

    init_inst.add_edge(special, extra, cap)
    init_inst.add_edge(extra, special, cap)

return (init_inst)
```

Listagem 3.2: Construção da instância inicial (retirado de [6])

```
compute_base_flow(graph, source, sink):
let base_flow[u][v] be the flow from u to v
let parent[u][t] be the pair (parent of u, parent type) in the path of type t found

for each edge (n,u) in graph.edges()
    base_flow[n][u] <- 0

while true
    (res, parent, sink_type) <- find_restricted_path(graph, source, sink, base_flow)
    if res = 0
        break
    else
        current_node <- sink
        type <- sink_type
        while cur_node ≠ source
            (prev_node, type) <- parent[cur_node][type]
            base_flow[prev_node][cur_node] <- base_flow[prev_node][cur_node] + res
            base_flow[cur_node][prev_node] <- base_flow[cur_node][prev_node] - res
            cur_node <- prev_node

return (base_flow)
```

Listagem 3.3: Cálculo do fluxo base (retirado de [6])

```
get_accessible_nodes(graph, flow, source):
let q be an empty deque of pairs (vertex, type)
let dq be an empty deque of pairs (vertex, type)
let parent[u][t] be the pair (parent of v, parent type) in the path of type t found
let seen[u] be true if a vertex has been seen with the normal type
let seen_up[u] be true if a vertex has been seen with the up type
let seen_all_types[u] be true if a vertex has been seen with any type

for each u in graph.vertices()
    seen[u] <- false
    seen_up[u] <- false
    seen_all_types[u] <- false

q.addFirst((source, Normal))
seen[source] <- true
seen_all_types[u] <- true
parent[source][Normal] <- (source, Normal)

while not q.is_empty()
    while not q.is_empty()
        (v, v_type) <- q.remove_first()
        for each u in graph.out_adjacent_nodes(v)
            if not seen[u]
                if graph.is_extra(u) and graph.get_special(u) ≠ v
                    and get_sp_type(flow, graph.get_special(u), u) = Zero
                    u_type <- Up
                else
                    u_type <- v_type

                if u_type = Normal or not seen_up[u]
                    (r, r_type) <- parent[v][v_type]
                    restricted_value <- min(graph.capacity(v,u) - flow[v][u],
                        restrict_flow(graph, flow, r, v, u))

                    if restricted_value > 0
                        parent[u][u_type] <- (v, v_type)
                        seen_all_types[u] <- true
                        if u_type = Normal
                            dq.add_first((u,u_type))
                            seen[u] <- true
                        else
                            dq.add_last((u, u_type))
                            seen_up[u] <- true
    //swap variables
    q <-> dq

return (seen_all_types)
```

Listagem 3.4: Obter nós queridos pela fonte (retirado de [6])

encontrar os nós queridos pela fonte, que pode ser encontrado na listagem 3.4, consiste em fazer uma pesquisa em largura a partir da fonte. Esta pesquisa verifica se, no caminho encontrado, o fluxo sobe por um nó extra onde o nó especial é do tipo Zero. Caso isto aconteça, dizemos que o tipo é Up e este nó é adicionado no fim da fila. Ao colocarmos um nó no fim da fila, fazemos com que os nós com tipo Normal sejam explorados antes dos nós com tipo Up. Assim, se existirem vários caminhos com o mesmo comprimento que tenham tipos diferentes, os caminhos com o tipo Up não serão explorados.

Após sabermos quais os nós que são acessíveis a partir da fonte, iremos criar a instância canónica. Para isto, temos que verificar se chegamos a alguns filhos do nó especial, a todos

```
build_canonical_instance(graph, access_nodes)
for each extra in graph.get_extra_nodes()
    sp <- graph.get_special(extra)
    let child_not_seen be an empty list of vertices
    children <- graph.get_children(sp)

    for each c in children
        if not access_nodes[c]
            child_not_seen.add(c)

    if child_not_seen.size() > 0 and child_not_seen.size() < children.size()

        cap <- graph.capacity(sp, extra)
        let extra' be a new extra vertex to be added to the graph
        graph.add_extra_node(extra', sp)

        graph.add_edge(sp, extra', cap)
        graph.add_edge(extra', sp,  cap)

        for each x in child_not_seen
            graph.remove_edge(extra, x)
            graph.remove_edge(x, extra)
            graph.add_edge(extra', x, cap)
            graph.add_edge(x, extra', cap)

return (graph)
```

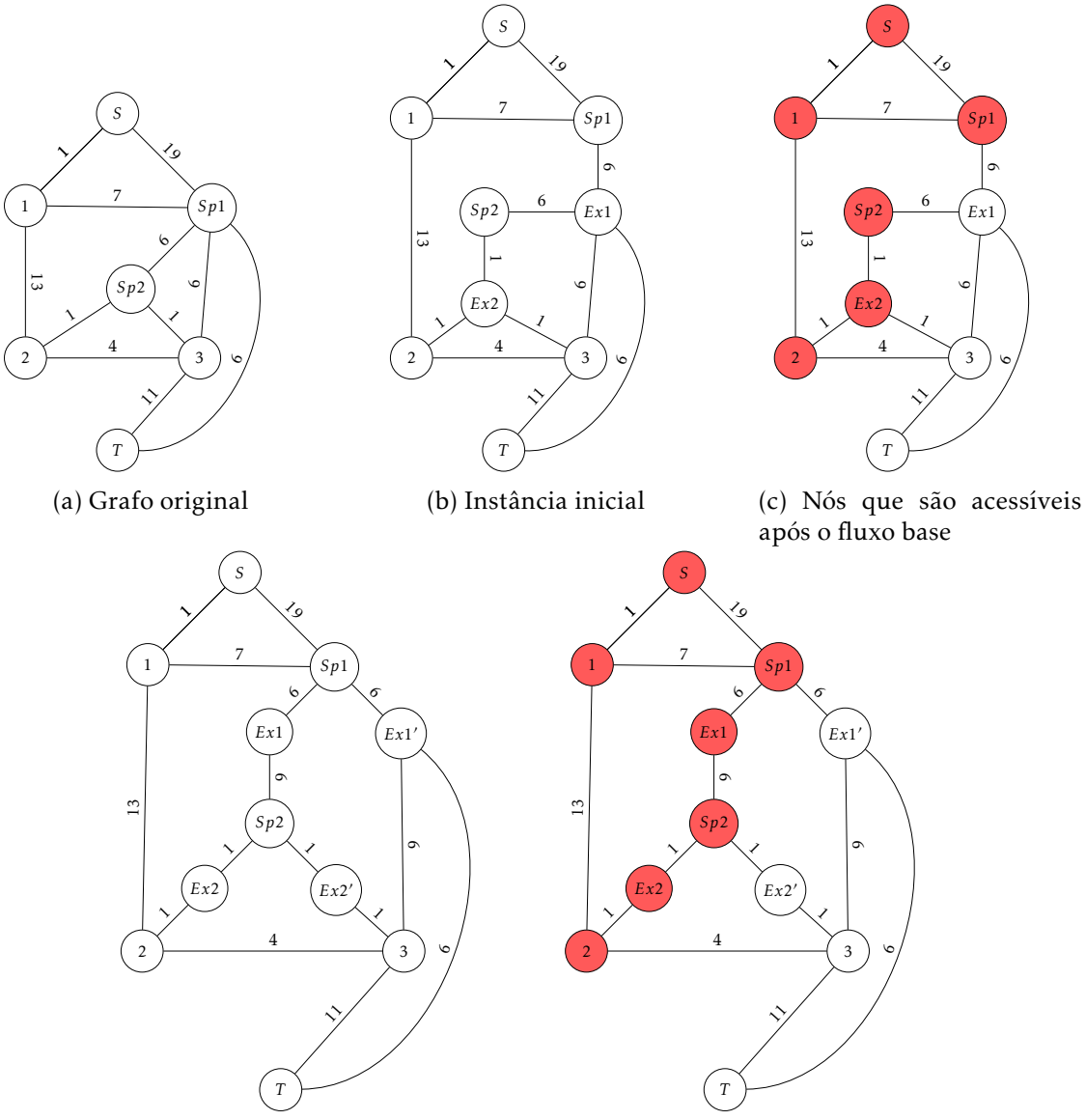
Listagem 3.5: Construir instância canónica (retirado de [6])

ou a nenhum. Se chegarmos a todos ou a nenhum nada é feito; caso contrário, iremos adicionar um segundo nó extra, como pode ser visto na listagem 3.5.

Como podemos ver no fim da listagem 3.1, o algoritmo termina após calcular o corte mínimo na instância canónica.

Na figura 3.1, podemos ver um exemplo da execução deste algoritmo. Começamos por apresentar a instância inicial do grafo original na figura 3.1b. Onde para cada nó especial do grafo original (figura 3.1a) foi adicionado um nó extra entre este nó e os seus filhos. A seguir, calculamos o fluxo base e apresentamos os nós que são acessíveis a partir da fonte, que estão assinalados a vermelho na figura 3.1c.

Por último, apresentamos a instância canónica, figura 3.1d, onde para cada um dos nós especiais foi adicionado mais um nó extra pois apenas conseguíamos chegar a alguns dos filhos dos nós especiais. Calculamos um fluxo máximo nesta instância e verificamos a quais nós conseguimos chegar a partir da fonte. Estes nós estão apresentados a vermelho na figura 3.1e. Portanto, temos que a capacidade do corte é $6 + 4 + 1 = 11$.



(a) Grafo original

(b) Instância inicial

(c) Nós que são acessíveis após o fluxo base

(d) Instância canónica

(e) Nós que são acessíveis após o fluxo na instância canónica

Figura 3.1: Exemplo da execução do algoritmo de corte mínimo com nós especiais

3.2 Algoritmo Proposto

Antes de apresentarmos o algoritmo proposto neste trabalho, iremos fazer algumas alterações (listagem 3.6) no algoritmo de corte mínimo com nós especiais [6], pois iremos precisar dos nós que são queridos pelos terminais após o fluxo base e após o fluxo na instância canónica. Agora, antes do retorno, começamos por passar o vector de booleanos dos nós acessíveis após o fluxo base para um vector que indica se um nó pertence ao conjunto da fonte ou do dreno. E o algoritmo passará a retornar um inteiro com a capacidade do corte mínimo; um vector que indica qual o conjunto a que o nó pertence após o fluxo base; e um vector que indica o conjunto a que o nó pertence após o fluxo na instância canónica.

```
special_min_cut2(graph, source, sink):
init_inst <- build_initial_instance(graph)
base_flow <- compute_base_flow(init_inst, source, sink)
access_base <- get_accesible_nodes(init_inst, base_flow, source)
can_inst <- build_canonical_instance(init_inst, access_base)
(part_final, cut_value)<- min_cut(can_inst, source, sink)

let part_base[u] be the terminal that wants node u

for each node in graph.vertices()
    if access_base[node]
        part_base[node] <- source
    else
        part_base[node] <- sink

return (part_base, part_final, cut_value)
```

Listagem 3.6: Alterações no algoritmo de corte mínimo com nós especiais

Após as alterações necessárias, apresentamos o algoritmo proposto na listagem 3.7. Este algoritmo consiste em duas grandes fases: a primeira onde encontraremos os nós queridos pelos nós terminais e a segunda onde serão resolvidas possíveis incompatibilidades e calculada a capacidade da divisão.

Primeira Fase:

A primeira fase consiste em executar o algoritmo explicado na secção anterior duas vezes. Executamos este algoritmo duas vezes, pois queremos saber quais são os nós queridos pelos dois terminais após o fluxo base e o fluxo na instância canónica serem calculados. Estes nós serão necessários para a fase seguinte. Para isto, temos que determinar qual dos dois terminais irá ser a fonte e o dreno em cada uma das execuções. Na primeira execução, teremos que a fonte é a fonte do grafo e o dreno é o dreno do grafo. Na segunda, iremos inverter os "papéis", ou seja, temos que a fonte é o dreno do grafo e o dreno é a fonte do grafo.

```
special_min_division(graph, s, t):
(part_baseS, part_finalS, cut_value) <- special_min_cut2(graph, s, t)
(part_baseT, part_finalT, cut_value_inv) <- special_min_cut2(graph, t, s)

(val_base, part_base, graph_base) <- phase2(graph, part_baseS, part_baseT, s, t)
(val_final, part_final, graph_final) <- phase2(graph, part_finalS, part_finalT, s, t)

if val_base < cut_value and val_base ≤ val_final
    return (part_base, val_base, graph_base)
else if val_final < cut_value
    return (part_final,val_final, graph_final)

return (part_finalS, cut_value, graph)
```

Listagem 3.7: Algoritmo de divisão mínima

Segunda Fase:

A segunda fase consiste em resolver possíveis incompatibilidades existentes e calcular a capacidade da divisão após as incompatibilidades serem resolvidas. Iremos resolver as incompatibilidades da seguinte forma: replicar os nós que são queridos pelos dois terminais; e determinar a qual dos terminais devem pertencer os nós que não são queridos por nenhum terminal.

Assim como a fase anterior, a segunda fase também será executada duas vezes. A diferença entre essas duas execuções está nos conjuntos de nós que iremos passar como argumento para essa função. Na primeira vez, iremos passar os nós queridos pela fonte e pelo dreno após ser calculado o fluxo base e, na segunda, iremos passar os nós queridos pela fonte e pelo dreno após ser calculado o fluxo na instância canónica.

```
phase2(graph, part_S, part_T, s, t):
(common_nodes,white_nodes,num_sets) <- get_incompatibilities(graph, part_S, part_T)
(replicated_graph, part_S) <- replicate_nodes(graph, common_nodes, part_S, s, t)
(groups) <- create_groups(replicated_graph, white_nodes, num_sets)

for each group in groups
    (part_S) <- get_minimal_cost(replicated_graph, group, part_S, s, t)
(division_value) <- get_division_value(replicated_graph, part_S)

return (division_value, part_S, replicated_graph)
```

Listagem 3.8: Segunda fase do algoritmo proposto

Como pode ser visto na listagem 3.8, a segunda fase começa com a verificação da existência de incompatibilidades entre os conjuntos de nós queridos pela fonte e pelo dreno. Esta verificação, descrita na listagem 3.9, é feita através de um ciclo, onde percorremos os nós do grafo e verificamos se os dois terminais querem aquele nó ou se nenhum terminal quer aquele nó. Se os dois quiserem aquele nó, então esse nó é adicionado à lista que irá conter todos os nós comuns e este nó é marcado como querido pelos dois terminais no vector que indica o número de conjuntos a que este nó pertence; se o nó não for querido

```
get_incompatibilities(graph, part_S, part_T, s, t):
let common_list be a list with common vertices
let white_nodes be a list with the vertices that none of the terminals want
let num_sets[u] be the number of sets of a vertex u

for each node in graph.vertices()
    if part_S[node] = s and part_T[node] = t
        common_list.add(node)
        num_sets[node] <- 2
    else if (part_S[node] = s or part_T[node] = t
        num_sets[node] <- 1
    else
        white_nodes.add(node)
        num_sets[node] <- 0

return (common_list, white_nodes, num_sets)
```

Listagem 3.9: Verificar possíveis incompatibilidades

por nenhum terminal, então este é adicionado à lista que irá conter todos os nós que não são queridos pela fonte nem pelo dreno e este nó é marcado como um nó não querido. No caso de ser querido apenas pela fonte ou pelo dreno, então o nó é marcado como querido por um terminal. Esta função irá retornar a lista com os nós comuns aos dois conjuntos, a lista com os nós que não pertencem a nenhum dos dois conjuntos e o vector com a indicação de a quantos conjuntos cada um dos nós pertence.

Após as incompatibilidades serem verificadas, iremos replicar os nós que sejam comuns aos dois terminais, caso estes existam. Como pode ser visto na listagem 3.10, começamos por percorrer a lista de nós comuns. Por cada um destes nós é criado um novo nó e garantimos que o novo nó irá fazer parte do conjunto do dreno. A seguir, iremos percorrer os nós que são antecessores diretos do nó que estamos a replicar. Se o nó antecessor tiver sido replicado, então iremos adicionar um arco entre o nó replicado do antecessor e o novo nó. Caso contrário, adicionamos um arco entre o nó antecessor e o novo nó. Depois, iremos percorrer os sucessores diretos do nó comum e verificar se o sucessor pertence ao conjunto do dreno. Se pertencer, então iremos remover o arco entre o nó comum e o sucessor e adicionar um arco entre o novo nó e o sucessor. No processo de replicação de nós, a ordem de tratamento dos nós é importante para obtermos o resultado esperado. Assumimos que se existe um arco (u, v) no grafo original, então u será tratado antes de v.

A seguir, serão criados grupos com os nós que não são queridos por nenhum terminal. Como apresentado na listagem 3.11, começamos com um ciclo onde iremos percorrer os nós que não são queridos por nenhum terminal. Caso o nó não tenha sido explorado, iremos criar um novo grupo e uma fila com este nó. A seguir, temos um ciclo que continua enquanto a fila não estiver vazia, onde iremos explorar os nós adjacentes do primeiro nó da fila. Caso o nó adjacente ainda não tenha sido explorado, marcamos este como explorado e verificamos se este também não é querido por nenhum terminal. Se não for, então adicionamos este nó ao grupo. Verificamos também se um dos nós é especial e se o outro é seu filho. Caso isto seja verdade, então iremos percorrer os filhos deste nó especial

```
replicate_nodes(graph, common_nodes, part, s, t):
let replicated_nodes[u] be the replica of vertex u if u was replicated

replicated_graph <- clone(graph)
for each n in graph.vertices()
    replicated_nodes[n] <- ∅

for each node in common_nodes
    let new_node be the replicated vertex
    replicated_graph.add_node(new_node)
    replicated_nodes[node] <- new_node
    part[new_node] <- t
    for each u in graph.in_adjacent_nodes(node)
        cap <- graph.capacity(u,node)
        if replicated_nodes[u] = ∅
            replicated_graph.add_edge(u, new_node, cap)
        else
            u' <- replicated_nodes[u]
            replicated_graph.add_edge(u', new_node, cap)
    for each u in graph.out_adjacent_nodes(node)
        if part[u] ≠ s
            cap <- graph.capacity(node, u)
            replicated_graph.remove_edge(node, u)
            replicated_graph.add_edge(new_node, u, cap)

return (replicated_graph, part)
```

Listagem 3.10: Replicação dos nós comuns

```
create_groups(graph, white_nodes, num_sets):
let seen[u] be true if u has been seen
let groups be a list with all groups that are created

for each node in graph.vertices()
    seen[node] <- false

for each w in white_nodes
    if not seen[w]
        let group be a new group
        let q be a queue
        seen[w] <- true
        group.add(w)
        q.enqueue(w)
        while not q.is_empty()
            x <- q.poll()
            for each v in graph.adjacent_nodes(x)
                if not seen[v]
                    seen[v] <- true
                    if num_sets[v] = 0
                        group.add(v)
                        q.enqueue(v)
                    if graph.is_special(v) and graph.is_child_of(x, v)
                        for each c in graph.get_children(v)
                            if not seen[c]
                                seen[c] <- true
                                if num_sets[c] = 0
                                    group.add(c)
                                    q.enqueue(c)
        groups.add(group)
return (groups)
```

Listagem 3.11: Criação de grupos com os nós que não são queridos por nenhum terminal

```
get_minimal_cost(graph, group, part, s, t):
let cost[u] be the cost of the group on terminal u
let seen[u][v] be the pair (terminal u, special v)

for each n in {s,t}
    for each u in graph.get_special_nodes()
            seen[n][u] <- false

cost[s] <- 0
cost[t] <- 0

for each node in group
    for each u in graph.adjacent_nodes(node)
        if not group.contains(u)
            terminal <- part[u]
            cap <- graph.capacity(node,u)
            if graph.is_special(node) and graph.is_child_of(u, node)
                if not seen[terminal][node]
                    seen[terminal][node] <- true
                    cost[terminal] -= cap
            else
                cost[terminal] -= cap

for each node in group
    if cost[s] < cost[t]
        part[node] <- s
    else
        part[node] <- t

return (part)
```

Listagem 3.12: Escolha do terminal para os nós que não são queridos

e verificar se existe mais algum filho que não seja querido por nenhum terminal. Caso exista algum filho, então adicionamos ao grupo. Quando a fila estiver vazia, adicionamos o grupo criado à lista de grupos. Repetimos este processo enquanto houver nós que não são queridos por nenhum terminal e que ainda não tenham sido explorados.

Conforme a listagem 3.8, a seguir à criação dos grupos, iremos calcular para cada grupo qual o terminal a que este deve pertencer de modo a conseguir o menor custo para a divisão. Como pode ser visto na listagem 3.12, começamos por percorrer os nós do grupo que estamos a calcular o custo. Para cada nó, iremos percorrer os nós que são adjacentes ao nó do grupo e verificar os seguintes casos: se um dos nós é especial e o outro é seu filho; ou se os dois nós são normais. No primeiro caso, verificamos se o arco entre o nó especial e o filho já foi contabilizado para aquele terminal; caso não tenha sido, então contabilizamos o arco ao custo deste terminal e marcamos como contabilizado. Para o caso dos dois nós serem normais, apenas contabilizamos o arco ao custo do terminal. Por fim, iremos verificar qual o terminal que irá produzir menor capacidade para a divisão e será a este terminal que os nós que não são queridos por nenhum terminal irão pertencer.

Por último, é calculada a capacidade da divisão. Como pode ser visto na listagem 3.13, começamos por percorrer os arcos do grafo e verificamos se os nós pertencem a conjuntos de terminais diferentes. Caso isto seja verdade, então iremos verificar se estamos a tratar de um arco entre um nó especial e seu filho ou um arco entre dois nós normais. Para o caso

```
get_division_value(graph, part):
let cost be the cost of the division
let seen_sp[v] be true if a special node v has been seen

cost <- 0

for each sp in graph.get_special_nodes()
    seen[sp] <- false

for each (u,v) in graph.edges()
    if part[u] ≠ part[v]
        if graph.is_special(u) and graph.is_child_of(v,u)
            if not seen_sp[u]
                cost += graph.capacity(u,v)
                seen_sp[u] <- true
        else
            cost += graph.capacity(u,v)

return (cost)
```

Listagem 3.13: Cálculo do custo da divisão

de ser um arco entre um nó especial e seu filho, temos que verificar se este nó especial já foi visto; caso não tenha sido, então o peso do arco irá fazer parte da capacidade da divisão e marcamos o nó especial como visto. Caso seja um arco entre dois nós normais, apenas contabilizamos este arco para a capacidade da divisão. Este processo é repetido enquanto houver arcos que ainda não tenham sido vistos.

Após as duas fases, o algoritmo proposto compara os resultados obtidos ao replicarmos os nós que são queridos pelos terminais após o fluxo base, após o fluxo na instância canónica e o resultado do algoritmo proposto em [6], escolhe a melhor capacidade e devolve o grafo e a partição que obtêm esta capacidade. Sempre que for devolvido um grafo onde houve replicação, isto significa que este grafo gerou melhor capacidade que o algoritmo proposto em [6].

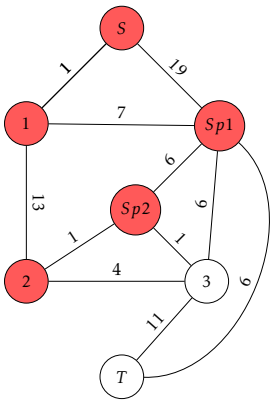
A seguir, iremos exemplificar uma execução deste algoritmo no mesmo grafo em que exemplificámos a execução do algoritmo de corte mínimo com nós especiais [6] na secção anterior. O exemplo pode ser visto na figura 3.2, onde começamos por executar o algoritmo apresentado em [6] duas vezes e obtemos os conjuntos dos nós queridos pela fonte e o dreno após o fluxo base (figuras 3.2a, 3.2b) e após o fluxo na instância canónica (figuras 3.2e e 3.2f) serem calculados.

A seguir, verificamos se existem incompatibilidades com os nós dos conjuntos da fonte e do dreno após o fluxo base. Como pode ser visto na figura 3.2c, os dois conjuntos têm o nó Sp2 em comum e não existe nenhum nó que não seja querido por nenhum dos terminais. Então iremos replicar o nó em comum. Ao replicarmos Sp2, temos que garantir que o novo nó, Sp2', tem os mesmos *inputs* de Sp2, por isso tivemos que adicionar o arco $(Sp1, Sp2')$. E adicionamos também um arco para cada um dos filhos de Sp2 que estejam no conjunto do dreno. No caso, só um filho está no conjunto do dreno, então adicionamos o arco $(Sp2', 3)$ e removemos o arco $(Sp2, 3)$ e obtemos o grafo da figura 3.2d. Neste grafo

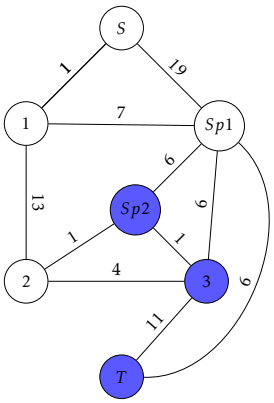
a capacidade desta divisão é $6 + 4 = 10$.

Agora iremos repetir o mesmo processo para os nós queridos após ser calculado o fluxo na instância canónica. Neste caso, não existem nós em comum entre os dois conjuntos, mas como pode ser visto na figura 3.2g, o nó 3 não é querido por nenhum dos dois conjuntos. Então iremos calcular a capacidade caso este nó faça parte do conjunto da fonte e do dreno. Ao analisarmos o grafo da figura 3.2h, onde o nó 3 faz parte do conjunto da fonte, temos que a capacidade da divisão é $11 + 6 = 17$. Ao fazermos o mesmo para o nó 3 a fazer parte do conjunto do dreno, temos que a capacidade é $6 + 4 + 1 = 11$. Comparando os dois resultados, temos que é melhor o nó 3 fazer parte do conjunto do dreno, pois teremos uma menor capacidade da divisão.

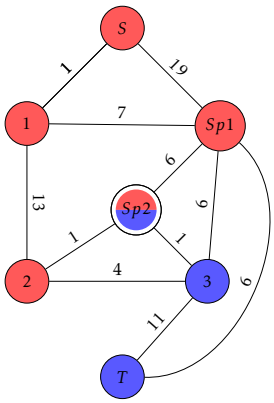
Por fim, iremos comparar os resultados obtidos pelas duas execuções da segunda fase do algoritmo e o resultado do algoritmo apresentado na secção 2.3.1. Para a execução com os nós obtidos através do fluxo base temos que a capacidade é 10. Para a execução com os nós obtidos através do fluxo na instância canónica temos que a capacidade é 11, assim como com o algoritmo de corte mínimo para nós especiais [6]. Como com os nós obtidos pelo fluxo base temos uma menor capacidade, então esta será a capacidade final da divisão.



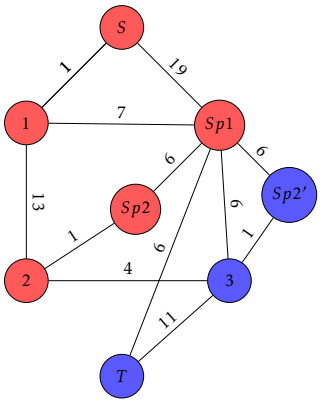
(a) Divisão da fonte com os nós do fluxo base



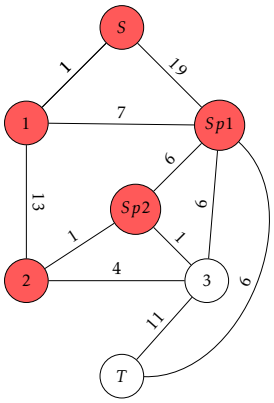
(b) Divisão do dreno com os nós do fluxo base



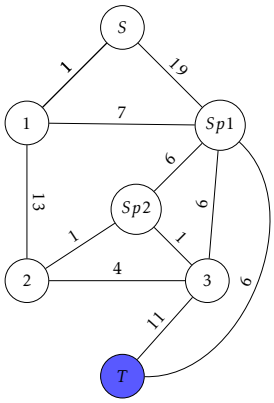
(c) União das divisões com os nós do fluxo base



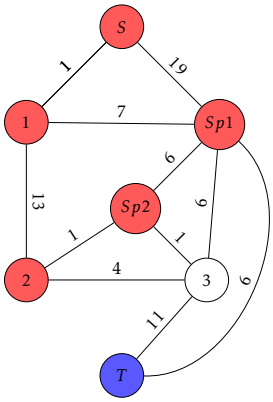
(d) Grafo com o Sp2 replicado



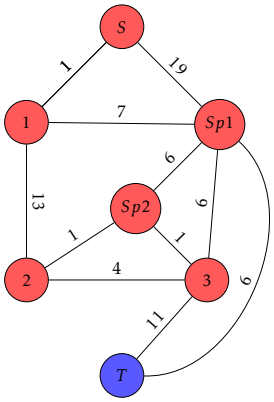
(e) Divisão da fonte com os nós após o fluxo na instância canónica



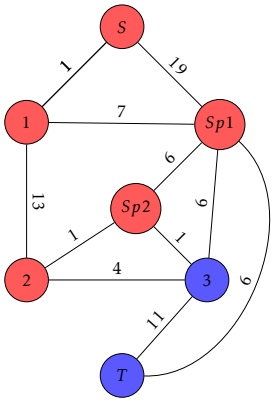
(f) Divisão do dreno com os nós após o fluxo na instância canónica



(g) União das divisões com os nós após o fluxo na instância canónica



(h) Nós não queridos a pertencer ao conjunto da fonte



(i) Nós não queridos a pertencer ao conjunto do dreno

Figura 3.2: Exemplo da execução do algoritmo de divisão mínima com nós especiais

3.3 Análise do Algoritmo

O algoritmo proposto tem duas fases. A primeira fase tem como complexidade duas vezes a complexidade do algoritmo de corte mínimo com nós especiais [6], $O(f_b|E| + |V||E|^2)$, onde f_b é o valor do fluxo base, $|V|$ é o número de nós e $|E|$ é o número de arcos.

A segunda fase está dividida em cinco etapas: verificar incompatibilidades, replicar nós comuns, criar grupos com os nós que não são queridos por nenhum terminal, calcular o custo dos grupos pertencerem ao conjunto de cada terminal e calcular o custo final da divisão.

A seguir apresentamos a complexidade de cada uma destas etapas:

- Verificar incompatibilidades: $O(|V|)$
- Replicar nós comuns: $O(|V|^2)$
- Criar grupos com os nós que não são queridos por nenhum terminal: $O(|V|^2)$
- Calcular o custo dos grupos pertencerem ao conjunto de cada terminal $O(|V|^2)$
- Calcular o custo final da divisão $O(|E|)$

Então a complexidade temporal do algoritmo proposto é $O(f_b|E| + |V||E|^2)$.

Capítulo 4

Divisão-k Mínima com Nós Especiais

4.1 Algoritmo de Corte-k Mínimo com Nós Especiais

Assim como no capítulo anterior, iremos começar por explicar um pouco mais detalhadamente o algoritmo de corte-k mínimo com nós especiais [6]. O algoritmo apresentado na secção 2.3.2 tem a estrutura da listagem 4.1. Onde começamos por calcular o corte

```
special_min_k_cut(graph, terminals):
let part[n] be the set of terminals that want n
let k_cut_value be the value of the k-cut

for each n in graph.vertices()
    part[n] <- ∅

for each source in terminals
    clone_graph <- clone(graph)

    let sink be a new vertex in the graph
    clone_graph.add_node(sink)
    for each t in terminals
        if t ≠ source
            clone_graph.add_edge(t, sink, +∞)

    (partition,cut_value) <- special_min_cut(clone_graph, source, sink)

    for each node in graph.vertices()
        if partition[node] = source
            part[node].add(source)

if has_incompatibilities(part)
    part <- solve_incompatibilities(graph, part)

k_cut_value <- get_k_cut_value(graph, part, terminals)

return(part, k_cut_value)
```

Listagem 4.1: Algoritmo de corte-k mínimo com nós especiais (retirado de [6])

mínimo com nós especiais para cada um dos k terminais. A seguir, verificamos a existência de incompatibilidades. Diz-se que existe uma incompatibilidade se existirem nós que mais do que um terminal quer ou nós que nenhum terminal quer, como pode ser visto na listagem 4.2.

```
1 has_incompatibilities(part):
2 for each p in part
3     if p.size() ≠ 1
4         return true
5 return false
```

Listagem 4.2: Verificação da existência de incompatibilidades (retirado de [6])

```
1 solve_incompatibilities(graph, part):
2 part <- solve_white_nodes(graph, part)
3 (groups, part) <- create_groups(graph, part)
4 for each group in groups
5     part <- solve_group(graph, part, group)
6 return (part)
```

Listagem 4.3: Resolver incompatibilidades existentes (retirado de [6])

Conforme apresentado na listagem 4.3, as incompatibilidades serão resolvidas em três etapas. A primeira etapa consiste em resolver o problema dos nós que não são queridos por nenhum terminal. Para resolver este problema, começamos por percorrer os nós do grafo até encontrar algum que não é querido por nenhum terminal. Adicionamos este nó à fila, ao grupo de nós que nenhum terminal quer e fazemos uma pesquisa em largura para encontrar nós adjacentes com o mesmo problema ou então para sabermos quais os terminais que fazem fronteira com este grupo. Diz-se que um terminal faz *fronteira* caso seja encontrado um nó adjacente que é querido por esse terminal. Quando a fila estiver vazia, associamos os terminais que fazem fronteira aos nós que pertencem ao grupo. O processo descrito na listagem 4.4 é repetido enquanto existirem nós que nenhum terminal quer.

A segunda etapa consiste em encontrar grupos de incompatibilidades, ou seja, unir nós adjacentes que são queridos por mais que um terminal e que tenham pelo menos um terminal em comum. No fim desta etapa, os grupos adjacentes e que tenham pelo menos um terminal em comum serão unidos. Conforme descrito na listagem 4.5, começamos por percorrer todos os nós do grafo até encontrarmos um nó que ainda não tenha sido explorado e que é querido por mais do que um terminal. Quando encontramos um nó com estas características, adicionamos este nó a uma fila, marcamos como explorado e criamos um novo grupo. Cada grupo tem um conjunto de nós que pertencem ao grupo e um conjunto de terminais que querem estes nós. A seguir, fazemos uma pesquisa em largura para encontrar outros nós que também sejam queridos por mais do que um terminal e que tenham pelo menos um terminal em comum. Quando a pesquisa acaba, iremos associar a

```
solve_white_nodes(graph, part):
for each node in graph.vertices()
    if part[node].is_empty()
        let seen[u] be true if vertex u has been seen
        let white_nodes be a list of vertices
        let q be a queue
        let terminals_in_border be a set of terminals

        seen[node] <- true
        q.enqueue(node)
        white_nodes.add(node)

        while not q.is_empty()
            v <- q.dequeue()
            for each u graph.adjacent_nodes(v)
                if not seen[u]
                    if part[u].is_empty()
                        seen[u] <- true
                        q.enqueue(u)
                        white_nodes.add(u)
                    else
                        terminals_in_border.add_all(part[u])
                        if graph.is_special(u) and graph.is_child_of(v,u)
                            seen[u] <- true
                            for each c in graph.get_children(u)
                                if not seen[c]
                                    seen[c] <- true
                                    if part[c].is_empty()
                                        q.enqueue(c)
                                        white_nodes.add(c)
                                    else
                                        terminals_in_border.add_all(part[c])
        for each w in white_nodes
            part[w].add_all(terminals_in_border)
return(part)
```

Listagem 4.4: Resolver o problema dos nós que não são queridos por nenhum terminal (retirado de [6])

cada nó do grupo os terminais encontrados e adicionar o grupo à lista de grupos.

Conforme mencionado anteriormente, iremos juntar grupos adjacentes e que tenham pelo menos um terminal em comum. Para isto, iremos seguir o processo apresentado na listagem 4.6, onde começamos por gerar todas as combinações de pares de grupos que tenham pelo menos uma ligação entre eles. Diz-se que existe uma *ligação* entre dois grupos se existir um arco entre eles ou se existirem filhos de um nó especial que pertençam aos dois grupos. A seguir, criamos uma estrutura *union find* e iniciamos um ciclo que se repete enquanto existirem grupos a serem unidos. Neste ciclo, iremos percorrer cada par de grupos e verificar se existe pelo menos um terminal em comum. Caso o par a ser tratado tenha um terminal em comum, então os grupos são unidos e o par é removido da lista de grupos a ser unidos.

A terceira etapa consiste em escolher o terminal a que cada grupo deve pertencer. Conforme descrito na listagem 4.7, começamos por percorrer os nós adjacentes de cada nó que pertence ao grupo e verificar se o nó adjacente é querido por apenas um terminal. Caso isto aconteça, então o arco entre estes dois nós faz parte da fronteira e o peso deste

```
create_groups(graph, part):
let groups be a list of groups where each group has a set of vertices and
     a set of terminals
let seen[u] be true if vertex u has been seen

for each node in graph.vertices()
    seen[node] <- false

for each node in graph.vertices()
    if not seen[node] and part[node].size() > 1
        let q be an empty queue of vertices
        q.enqueue(node)
        seen[node] <- true

        let group be a new group
        group.nodes.add(node)
        group.terminals.add_all(part[node])

        while not q.is_empty()
            v <- q.dequeue()
            for each u in graph.adjacent_nodes(v)
                if not seen[u]
                    if part[u].size() > 1 and
                        group.terminals.contains_any(part[u])
                        q.enqueue(u)
                        seen[u] <- true
                        group.nodes.add(u)
                        group.terminals.add_all(part[u])
                    else if graph.is_special(u) and graph.is_child_of(v,u)
                        for each s in graph.get_children(u)
                            if not seen[s] and part[s].size() > 1 and
                                group.terminals.contains_any(part[s])
                                q.enqueue(s)
                                seen[s] <- true
                                group.nodes.add(s)
                                group.terminals.add_all(part[s])
                    else if part[u].size() = 1 and not graph.is_special(u)
                        seen[u] <- true
        for each n in group.nodes
            part[n] <- group.terminals
        groups.add(group)

if groups.size() > 1
    groups <- join_groups(graph,groups)
return (groups, part)
```

Listagem 4.5: Encontrar grupos de incompatibilidades (retirado de [6])

```
join_groups(graph, groups):
let join_list be a list of pairs of integers (group1,group2)

for each g1 in [0, groups.size() -2]
    for each g2 in [g1 +1, groups.size() -1]
        if hasEdgeBetween(graph, groups[g1], groups[g2])
            join_list.add((g1,g2))
let part be a union find data structure with size of groups

changes <- true

while changes and not join_list.is_empty()
    changes <- false

    for each (g1,g2) in join_list
        f1 <- part.find(g1)
        f2 <- part.find(g2)
        if f1 = f2
            join_list.remove((g1,g2))
        else if groups[f1].terminals.contains_any(groups[f2].terminals)
            part.union(f1,f2)
            parent <- part.find(f1)
            if f1 = parent
                other <- f2
            else
                other <- f1
            groups[parent].terminals.add_all(groups[other].terminals)
            groups[parent].nodes.add_all(groups[other].nodes)
            groups[other] <- ∅

            join_list.remove((g1,g2))
            changes <- true
groups.remove_all(∅)
return(groups)




hasEdgeBetween(graph, g1, g2):
    for each v in g1.nodes
        for each u in graph.adjacent_nodes(v)
            if g2.nodes.contains(u)
                return true
            if graph.is_special(u) and graph.is_child_of(v,u)
                for each s in graph.get_children(u)
                    if g2.nodes.contains(s)
                        return true
return false
```

Listagem 4.6: Unir grupos que tenham pelo menos um terminal em comum (retirado de [6])

```
solve_group(graph, part, group):
let cost_terminal[t] be the cost gain if a terminal is chosen for the group
let seen[t][s] be true if a child of special vertex s belongs to terminal t

for each t in group.terminals
    for each sp in graph.get_special_nodes()
        seen[t][sp] <- false

for each v in group.nodes
    for each w in graph.adjacent_nodes(v)
        if part[w].size() = 1
            tw <- part[w].getElement()
            if tw in group.terminals
                if graph.is_special(v) and graph.is_child_of(w,v)
                    if not seen[tw][v]
                        seen[tw][v] <- true
                        cost_terminal[tw] += graph.capacity(v,w)
                else if graph.is_special(w) and graph.is_child_of(v,w)
                    if not seen[tw][w]
                        seen[tw][w] <- true
                        cost_terminal[tw] += graph.capacity(v,w)
                else
                    cost_terminal[tw] += graph.capacity(v,w)
        if graph.is_special(w) and graph.is_child_of(v, w)
            for each u in graph.get_children(w)
                if part[u].size() = 1
                    tu <- part[u].getElement()
                    if tu in group.terminals and not seen[tu][w]
                        seen[tu][w] <- true
                        cost_terminal[tu] -= graph.capacity(v,w)
max <- 0
for each t in group.terminals
    if cost_terminal[t] > max
        max <- cost_terminal[t]
        max_terminal <- t

for each v in group.nodes
    part[v].remove_all()
    part[v].add(max_terminal)

return(part)
```

Listagem 4.7: Escolher o melhor terminal para um grupo pertencer (retirado de [6])

arco deverá ser contabilizado para o custo deste terminal. Existem três situações possíveis para estes dois nós: o nó que pertence ao grupo é um nó especial e o nó adjacente é seu filho; o nó adjacente é um nó especial e o nó que pertence ao grupo é seu filho; os dois nós são normais. Para o caso do nó adjacente ser o nó especial e o nó que pertence ao grupo ser seu filho, teremos que percorrer os filhos do nó especial e contabilizar estes arcos caso ainda não tenham sido contabilizados. No caso do nó que pertence ao grupo ser especial e o adjacente ser seu filho, verificamos se este arco já foi contabilizado, e caso não tenha sido, iremos contabilizar e marcar como contabilizado. No caso de os dois nós serem normais, apenas contabilizamos o arco entre os dois nós que estamos a verificar. Após todos os nós terem sido percorridos, iremos verificar qual o terminal que irá produzir a maior redução na capacidade do corte e este será o terminal escolhido. Para finalizar esta etapa, iremos associar aos nós deste grupo o terminal escolhido.

Por último será calculada a capacidade do corte-k com nós especiais [6]. Conforme

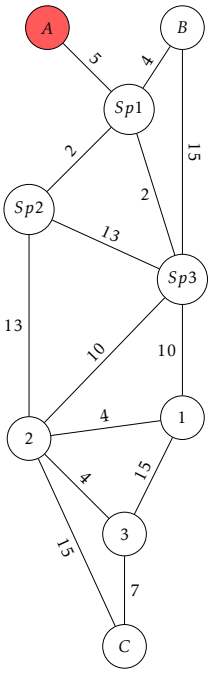
```
get_k_cut_value(graph, part, terminals):
let seen[t][s] be true if a child of special vertex s belongs to the set of terminal t

for each t in terminals
    for each sp in graph.get_special_nodes()
        seen[t][sp] <- false

cut_k_value <- 0
for each (u,v) in graph.edges()
        t_u <- part[u].getElement()
        t_v <- part[v].getElement()
        if t_u ≠ t_v
            if graph.is_special(u) and graph.is_child_of(v, u)
                if not seen[t_v][u]
                    seen[t_v][u] <- true
                    cut_k_value += graph.capacity(u,v)
            else if graph.is_special(v) and graph.is_child_of(u, v)
                if not seen[t_u][v]
                    seen[t_u][v] <- true
                    cut_k_value += graph.capacity(u,v)
            else
                cut_k_value += graph.capacity(u,v)
return (cut_k_value)
```

Listagem 4.8: Cálculo da capacidade do corte-k (retirado de [6])

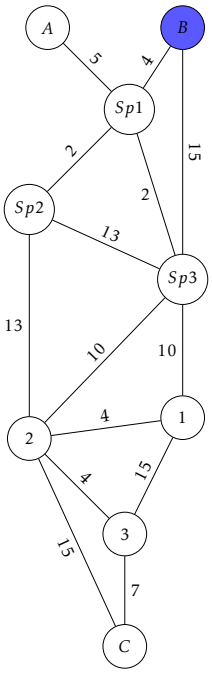
descrito na listagem 4.8, começamos por percorrer os arcos do grafo e verificar se os dois nós pertencem ao conjunto do mesmo terminal. Caso não seja verdade, iremos verificar se estamos a tratar de um arco entre um nó especial e seu filho ou de um arco entre dois nós normais. Se estivermos a tratar de um arco entre um nó especial e seu filho, temos que verificar se este nó já foi visto. Caso não tenha sido visto, então temos que contabilizar este arco e marcar como visto. Se estivermos a tratar de um arco entre nós normais, apenas iremos contabilizar o arco para a capacidade do corte. Após ser calculada a capacidade do corte-k, o algoritmo principal (listagem 4.1) retorna a capacidade do corte e a partição que irá produzir esta capacidade.

Na figura 4.1 podemos ver um exemplo de uma execução deste algoritmo. Começamos por apresentar os nós queridos por cada terminal após o cálculo do corte mínimo nas figuras 4.1a, 4.1b, 4.1c.

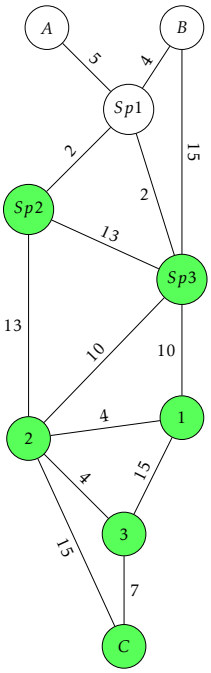
A seguir, iremos unir os cortes de todos os terminais e verificar se existe alguma incompatibilidade, ou seja, encontrar nós que não são queridos por nenhum terminal ou nós que são queridos por mais do que um terminal. Como pode ser visto na figura 4.1d, o nó $Sp1$ não é querido por nenhum terminal. Então iremos criar um grupo com este nó e verificar quais os terminais que fazem fronteira. Ao observarmos a figura 4.1e, podemos ver que o nó $Sp1$ faz fronteira com todos os terminais. Então iremos verificar o valor que será reduzido caso este nó pertença ao conjunto de cada um dos terminais. Temos que estes valores são: 5 para o terminal A; 4 para o terminal B; 2 para o terminal C. Como o terminal A irá produzir uma maior redução na capacidade do corte, então este é o terminal escolhido. O corte final deste grafo está apresentado na figura 4.1f e tem capacidade $15 + 4 + 2 = 21$.



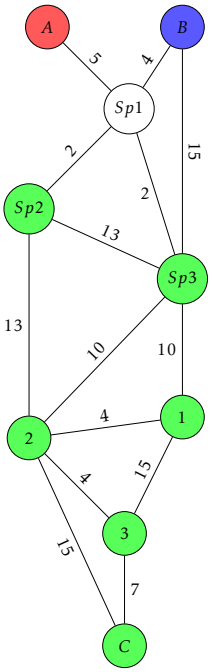
(a) Nós queridos pelo terminal A



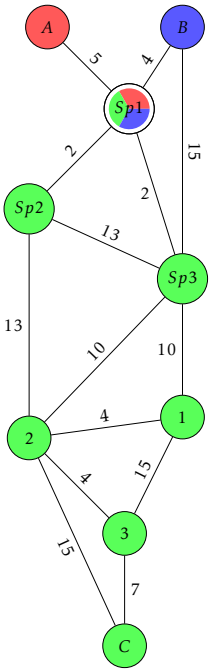
(b) Nós queridos pelo terminal B



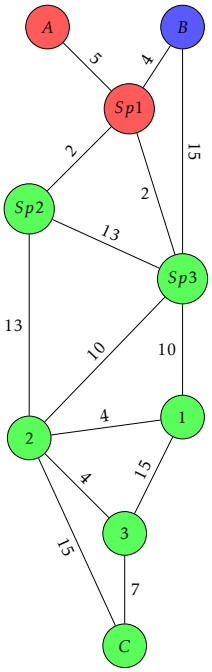
(c) Nós queridos pelo terminal C



(d) União dos cortes



(e) Grafo após tratar do nó que nenhum terminal quer



(f) Corte final

Figura 4.1: Exemplo da execução do algoritmo de corte-k mínimo com nós especiais

4.2 Algoritmo Proposto

O algoritmo proposto tem a estrutura apresentada na listagem 4.9, onde começamos por calcular o corte mínimo com nós especiais para cada um dos k terminais. Assim como no algoritmo proposto no capítulo anterior, também iremos usar o algoritmo de corte mínimo com nós especiais com as alterações apresentadas na secção 3.2, pois os nós que são queridos por cada um dos terminais após o fluxo base e o fluxo na instância canónica serão utilizados para resolver as possíveis incompatibilidades existentes ao unir os cortes mínimos de cada terminal.

```
special_min_k_division(graph, terminals):
let part_base[n] be the set of terminals that want n
let part_final[n] be the set of terminals that want n
for each n in graph.vertices()
    part_base[n] <- ∅
    part_final[n] <- ∅

for each source in terminals
    clone_graph <- clone(graph)
    let sink be a new node in the graph
    clone_graph.add_node(sink)
    for each t in terminals
        if t ≠ source
            clone_graph.add_edge(t, sink, +∞)
    (part_b, part_f, cut_value) <- special_min_cut2(clone_graph, source, sink)
    for each node in graph.vertices()
        if part_b[node] = source
            part_base[node].add(source)
        if part_f[node] = source
            part_final[node].add(source)

if has_incompatibilities(part_base)
    (graph_base, part_base, base_rep) <- replicate(graph, part_base, terminals)
    part_base <- solve_incompatibilities(graph, part_base)
    val_base <- get_k_cut_value(graph_base, part_base)

if has_incompatibilities(part_final)
    (graph_final, part_final, final_rep) <- replicate(graph, part_final, terminals)
    part_final <- solve_incompatibilities(graph_final, part_final)
    val_final <- get_k_cut_value(graph_final, part_final)
    if final_rep
        part_k_cut <- solve_incompatibilities(graph, part_final)
        val_k_cut <- get_k_cut_value(graph, part_k_cut)
    else
        part_k_cut <- part_final
        val_k_cut <- val_final

if val_base < val_k_cut and val_base ≤ val_final
    return (part_base, val_base, graph_base)
else if val_final < val_k_cut
    return (part_final, val_final, graph_final)
return (part_k_cut, val_k_cut, graph)
```

Listagem 4.9: Algoritmo de divisão-k mínima

Após os k cortes serem calculados, iremos verificar a existência de incompatibilidades, ou seja, iremos procurar por nós que não são queridos por nenhum terminal ou nós que são queridos por mais do que um terminal.

Se existirem incompatibilidades, então iremos começar por replicar os nós queridos por mais que um terminal. Como pode ser visto no início da listagem 4.10, começamos por encontrar estes nós. Para isto, iremos percorrer os nós do grafo e verificar se estes são queridos por mais que um terminal. Caso isto seja verdade, adicionamos este nó à lista que irá conter todos os nós que são queridos por mais que um terminal. A seguir, iremos percorrer os terminais que querem este nó e para cada um destes terminais, iremos criar um novo nó e marcamos que o novo nó pertence a este terminal. Após termos os nós que são queridos por mais que um terminal, então iremos replicá-los. Começamos por percorrer os nós queridos por mais que um terminal e por cada nó iremos percorrer os terminais. Caso este nó tenha sido replicado neste terminal, então iremos percorrer os nós adjacentes ao nó comum. Para os antecessores diretos do nó comum, iremos adicionar um arco entre o antecessor e o novo nó, caso o antecessor não tenha sido replicado neste terminal. Caso o antecessor tenha sido replicado neste terminal, então iremos adicionar um arco entre o nó replicado do antecessor e o novo nó. Para os sucessores diretos do nó comum, iremos remover o arco entre o nó comum e o sucessor e adicionar um arco entre o novo nó e o sucessor, caso o sucessor pertença a este terminal. Caso o sucessor tenha sido replicado, então iremos adicionar um arco entre o novo nó e o nó replicado do sucessor. O algoritmo de replicação retorna o novo grafo, um vector que indica a que terminal os nós pertencem e um booleano a dizer se houve replicação ou não. Assim como no algoritmo de divisão mínima, a ordem pela qual os nós são tratados é relevante para conseguirmos o resultado esperado. Assumimos que se existe um arco (u, v) no grafo original, então u será tratado antes do que v.

Após a replicação, iremos resolver os restantes problemas de incompatibilidades conforme descrito na secção anterior (listagem 4.3). Começamos pelos nós que não são queridos por nenhum terminal. Procuramos por todos estes nós e verificamos quais os terminais que fazem fronteira com estes nós. A seguir, serão criados grupos com nós que são queridos por mais do que um terminal. Os grupos criados vão ser unidos, caso existam grupos adjacentes. E para cada grupo, é calculado o valor a ser reduzido da capacidade da divisão, caso este grupo faça parte do conjunto de cada terminal. O terminal escolhido será o que produzir maior redução na capacidade. Por fim, é calculada a capacidade da divisão.

Como pode ser visto na listagem 4.9, o processo de replicação e resolução de incompatibilidades é feito duas vezes, uma vez com os nós queridos pelos terminais após o fluxo base e a outra com os nós queridos após o fluxo na instância canónica. Se houver replicação com os nós queridos após o fluxo na instância canónica, então teremos que resolver as incompatibilidades e calcular a capacidade do corte-k sem replicação. Caso contrário, a capacidade do corte-k irá ser igual à da divisão-k com os nós após o fluxo na instância canónica.

```
replicate_nodes(graph, part, terminals):

(common_nodes, replicated, new_part) <- get_common_nodes(graph, part, terminals)

if common_nodes.isEmpty()
    return (graph, part, false)

replicated_graph <- clone(graph)

for each node in common_nodes
    for each t in terminals
        if replicated[node][t] ≠ ∅
            new_node <- replicated[node][t]
            for each n in graph.in_adjacent_nodes(node)
                cap <- graph.capacity(n, node)
                if replicated[n][t] = ∅
                    replicated_graph.add_edge(n, new_node, cap)
                else
                    n' <- replicated[n][t]
                    replicated_graph.add_edge(n', new_node, cap)
            for each n in graph.out_adjacent_nodes(node)
                cap <- graph.capacity(node, n)
                if new_part[n] = t
                    replicated_graph.remove_edge(node, n)
                    replicated_graph.add_edge(new_node, n, cap)
                else if replicated[n][t] ≠ ∅
                    n' <- replicated_graph[n][t]
                    replicated_graph.add_edge(new_node, n', cap)

return (replicated_graph, new_part, true)




get_common_nodes(graph, part, terminals):
let common_nodes be a list with common vertices
let replicated[n][t] be not empty if n has been replicated on t
let new_part[n] be the terminal that wants n

for each i in graph.vertices()
    for each t in terminals
        replicated[i][t] <- ∅
    new_part[i] <- part[i]

for each node in graph.vertices()
    if part[node].size() > 1
        new_part[node] <- part[node].removeFirst()
        common_nodes.add(node)
        for each t in part[node]
            let new_node be the new vertex
            replicated[node][t] <- new_node
            new_part[new_node] <- t

return(common_nodes, replicated, new_part)
```

Listagem 4.10: Replicação dos nós queridos por mais que um terminal

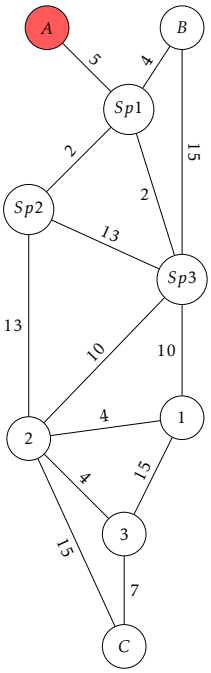
Por último, o algoritmo compara os valores obtidos pela execução com os nós queridos pelos terminais após o fluxo base, após o fluxo na instância canónica e o corte-k mínimo com nós especiais [6] e a melhor capacidade é a escolhida.

Na figura 4.2, iremos exemplificar uma execução do algoritmo proposto com o mesmo grafo com que exemplificámos o algortimo de corte-k mínimo com nós especiais na secção anterior. Começamos por calcular o corte mínimo para cada um dos k terminais e obtemos os nós queridos por cada um dos terminais após o fluxo base (figuras 4.2a, 4.2b e 4.2c) e o fluxo na instância canónica (figuras 4.1a, 4.1b e 4.1c).

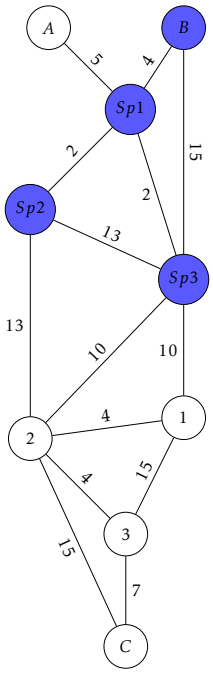
A seguir, iremos verificar se existem nós queridos por mais que um terminal após o fluxo base. Como vemos na figura 4.2d, o nó Sp2 é querido pelo terminal B e C. Então este nó será replicado. Ao replicarmos Sp2, temos que garantir que o novo nó, $Sp2'$, irá ter os mesmos *inputs* de Sp2, por isso, o arco $(Sp1, Sp2')$ foi adicionado. Temos também que verificar quais os filhos de Sp2 que pertencem ao conjunto do terminal C. Neste caso, apenas o nó 2 pertence ao conjunto do terminal C. Então iremos remover o arco $(Sp2, 2)$ e adicionar o arco $(Sp2', 2)$. Como não temos nenhum nó que não seja querido por nenhum terminal, então o grafo da figura 4.2e é o grafo obtido ao considerarmos os nós queridos pelo fluxo base, cuja capacidade é $5 + 2 + 10 = 17$.

Conforme descrito anteriormente, iremos repetir o mesmo processo com os nós queridos após o fluxo na instância canónica. Como pode ser visto na figura 4.1d, ao unirmos os cortes obtidos por cada um dos terminais, temos que o nó Sp1 não é querido por nenhum dos terminais e nenhum nó é querido por mais que um terminal. Por isso, não irá haver replicação e o algoritmo a ser usado é o algoritmo de corte-k mínimo com nós especiais [6] (exemplificado na figura 4.1).

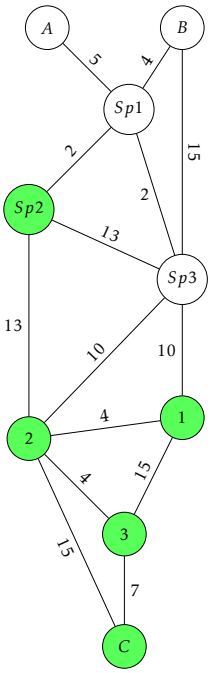
Por fim, o algoritmo irá comparar a capacidade obtida pelos nós queridos após o fluxo base, após o fluxo na instância canónica e o algoritmo proposto em [6]. Para os nós queridos pelo fluxo base a capacidade é 17. Para os nós queridos após o fluxo na instância canónica e o algoritmo de corte mínimo com nós especiais a capacidade é 21. Como para os nós queridos pelo fluxo base obtemos uma melhor capacidade, então este será escolhido pelo algoritmo.



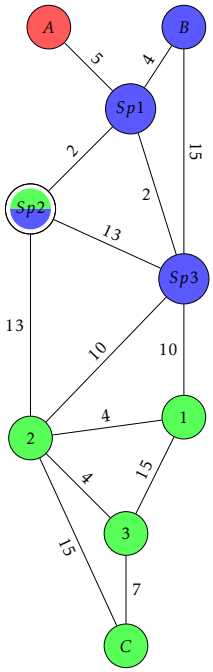
(a) Nós queridos pelo terminal A após o fluxo base



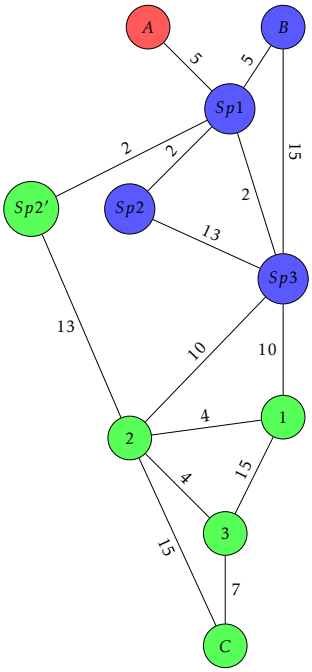
(b) Nós queridos pelo terminal B após o fluxo base



(c) Nós queridos pelo terminal C após o fluxo base



(d) União dos nós queridos pelos terminais após o fluxo base



(e) Grafo com nó Sp2 replicado

Figura 4.2: Exemplo da execução do algoritmo de divisão-k mínima com nós especiais

4.3 Análise do Algoritmo

O algoritmo proposto irá ter a complexidade do algoritmo de corte-k mínimo com nós especiais [6], $O(kf_b|E|+|V|^4)$, onde f_b é o máximo dos valores dos fluxos máximos, $|V|$ é o número de nós e $|E|$ é o número de arcos, juntamente com a complexidade da replicação.

Para a replicação, temos que analisar a complexidade de criar os nós replicas e a complexidade de replicar os nós queridos por mais que um terminal. A seguir apresentamos estas complexidades:

- Criar os nós replica (isolados): $O(k|V|)$
- Replicar os nós queridos por mais que um terminal: $O(k|E|+|E||V|)$

Logo a complexidade da replicação é $O(k|E|+|E||V|)$. Como $(k|E|+|E||V|) < |V|^4$, então a complexidade do algoritmo proposto é $O(kf_b|E|+|V|^4)$.

Capítulo 5

Resultados Experimentais

5.1 Conjunto de Teste

Neste capítulo iremos apresentar os resultados obtidos pelos algoritmos de divisão mínima com nós especiais e divisão-k mínima com nós especiais e compará-los com os algoritmos de corte mínimo com nós especiais e corte-k mínimo com nós especiais, respectivamente.

Primeiramente, iremos apresentar as características dos grafos gerados aleatoriamente que fazem parte do conjunto de teste. Para gerar os grafos, tivemos em consideração as características que representam o problema inicial, ou seja, grafos orientados, fracamente conexos e acíclicos. Para além destas características, os grafos também têm as características que estão resumidas na tabela 5.1 para os 100.310 grafos "pequenos" e na tabela 5.2 para os 20.000 grafos "grandes".

Tabela 5.1: Características dos grafos pequenos

	Mínimo	Máximo	Média
Número de nós	7	60	32,9
Número de terminais	3	9	6,8
Número de nós especiais	1	6	3,5
Número de arcos	8	133	58

Tabela 5.2: Características dos grafos grandes

	Mínimo	Máximo	Média
Número de nós	1.000	1.000	1.000
Número de terminais	10	20	15,0
Número de nós especiais	50	100	74,9
Número de arcos	1.904	12.801	3.889,5

Para testar o algoritmo de divisão mínima gerámos grafos com dois terminais a partir dos grafos usados para testar o algoritmo de divisão-k mínima. Em cada um dos grafos com k terminais, fixámos o dreno no último nó t do grafo e considerámos as fontes $T\backslash\{t\}$. Logo, para cada grafo com k terminais temos $k-1$ grafos com 2 terminais. Os 100.310 grafos pequenos com k terminais geraram 575.685 grafos com 2 terminais. Enquanto que os 20.000 grafos grandes com k terminais geraram 156.085 grafos grandes com 2 terminais.

Para comparar os resultados obtidos pelos algoritmos, iremos usar uma tabela que irá comparar subconjuntos do conjunto de algoritmos, ou seja, indicará quais os algoritmos do subconjunto que obtiveram melhor resultado do que os restantes algoritmos, excluindo o conjunto vazio. Também apresentaremos uma tabela com os tempos médios de execução dos algoritmos. Além destas tabelas, iremos fazer um estudo mais detalhado dos grafos onde ocorreu a replicação. Para estes grafos, iremos apresentar uma nova tabela a comparar os subconjuntos de algoritmos, uma tabela a apresentar o tipo dos nós especiais que foram replicados e uma tabela a apresentar a relação entre os nós normais e especiais que foram replicados.

Todos os algoritmos que iremos comparar foram implementados em Java. Os algoritmos de corte mínimo e corte-k mínimo concebidos para nós especiais foram cedidos pelo Ricardo Martins. Os testes foram executados numa máquina na Amazon WebServices (r5d.2xlarge) com 64 GBytes de RAM, CPU Intel Xeon Platinum série 8000 3,1 GHertz e a usar a Java VM versão 1.8.

5.2 Divisão Mínima com Nós Especiais

Para avaliar o algoritmo de divisão mínima com nós especiais, iremos comparar os seguintes algoritmos:

- O algoritmo de divisão mínima com os nós queridos pelo fluxo base (designado pela letra B);
- O algoritmo de divisão mínima com os nós queridos após o fluxo na instância canónica (designado pela letra C);
- O algoritmo de corte mínimo com nós especiais (designado pela letra A), proposto em [6].

As tabelas 5.3 e 5.4 apresentam os resultados obtidos ao comparar os três algoritmos para os grafos pequenos e grandes, respectivamente. Podemos observar que para os grafos pequenos, os três algoritmos calculam o melhor resultado em 93,6% dos casos. Enquanto que para os grafos grandes, os três algoritmos obtêm o melhor resultado em 91,5% dos casos. Também podemos notar que em menos de 0,1% dos grafos pequenos e grandes, o algoritmo de corte mínimo com nós especiais foi melhor que os restantes algoritmos.

Tabela 5.3: Comparação dos algoritmos de divisão mínima e corte mínimo com nós especiais para os grafos pequenos

Algoritmo	# Grafos	% Grafos
A	196	<0,1
B	8.037	1,4
C	606	0,1
AB	612	0,1
BC	437	<0,1
AC	27.163	4,7
ABC	538.634	93,6

Tabela 5.4: Comparação dos algoritmos de divisão mínima e corte mínimo com nós especiais para os grafos grandes

Algoritmo	# Grafos	% Grafos
A	22	<0,1
B	6.182	3,9
C	312	0,2
AB	3	<0,1
BC	67	<0,1
AC	6.756	4,3
ABC	142.743	91,5

As tabelas 5.5 e 5.6 apresentam os tempos totais, em segundos, que os algoritmos de divisão mínima com nós especiais (listagem 3.7) e corte mínimo com nós especiais demoraram a calcular a divisão para todos os grafos pequenos e grandes, respectivamente. Podemos observar que, para os grafos pequenos, o algoritmo de divisão mínima é 33% mais lento do que o algoritmo de corte mínimo, enquanto que para os grafos grandes, o algoritmo de divisão é 2,2 vezes mais lento do que o de corte mínimo.

Tabela 5.5: Tempos totais de execução dos algoritmos de divisão mínima e corte mínimo com nós especiais para todos os grafos pequenos

Algoritmo	Tempo (s)
Divisão Mínima com Nós Especiais	40,741
Corte Mínimo com Nós Especiais	30,684

Tabela 5.6: Tempos totais de execução dos algoritmos de divisão mínima e corte mínimo com nós especiais para todos os grafos grandes

Algoritmo	Tempo (s)
Divisão Mínima com Nós Especiais	1.070,304
Corte Mínimo com Nós Especiais	490,302

Análise dos Grafos com Replicação

Ao correr o algoritmo de divisão mínima com nós especiais para os grafos pequenos, obtivemos 8.073 grafos replicados. A tabela 5.7 compara os algoritmos B e C apenas nestes grafos. Ao analisar esta tabela, podemos observar que em 97,8% dos casos onde ocorreu replicação, os nós utilizados são os queridos após o fluxo base.

Tabela 5.7: Comparação dos algoritmos de divisão mínima para os grafos pequenos onde houve replicação

Algoritmo	# Grafos	% Grafos
B	7.787	96,5
C	180	2,2
BC	106	1,3

Dos 8.073 grafos pequenos onde houve replicação, 3.617 grafos replicaram apenas nós especiais e 4.456 grafos replicaram nós especiais e nós normais. A seguir, apresentamos a tabela 5.8 que indica o tipo do nó especial nos dois terminais que querem o nó a ser replicado. Como podemos observar, existem 947 nós especiais que são do tipo Up nos dois terminais que querem estes nós e 2.292 nós especiais que são Up em um terminal e Down em outro terminal.

Tabela 5.8: Tipo dos nós especiais que foram replicados nos grafos pequenos

Tipo	# Nós	% Nós
Up	947	6,7
Down	18	0,1
Zero	4.479	31,5
Zero Up	6.139	43,2
Zero Down	337	2,4
Up Down	2.292	16,1

A tabela 5.9 apresenta a relação dos nós normais com os nós especiais que foram replicados. Ao analisar esta tabela, podemos notar que nunca houve nós normais replicados isolados de nós especiais. Assim como podemos notar que existem 2.066 grafos onde os nós normais replicados são pais dos nós especiais replicados e apenas 10 grafos onde o nó normal é irmão do nó especial.

Tabela 5.9: Relação entre os nós normais e especiais que foram replicados nos grafos pequenos

Relação	# Grafos	% Grafos
Pais	2.066	46,4
Filhos	32	0,7
Irmãos	10	0,2
Pais e filhos	2.347	52,7

A tabela 5.10 apresenta a comparação dos algoritmos B e C apenas nos 6.217 grafos grandes onde houve replicação. Podemos observar que os nós queridos pelo fluxo base não são utilizados em apenas 0,4% dos grafos.

Tabela 5.10: Comparação dos algoritmos de divisão mínima para os grafos grandes onde houve replicação

Algoritmo	# Grafos	% Grafos
B	6.182	99,4
C	25	0,4
BC	10	0,2

Dos 6.217 grafos grandes onde houve replicação, 5.431 grafos replicaram apenas nós especiais e 786 replicaram nós normais e especiais. A seguir apresentamos a tabela 5.11 com os tipos dos nós especiais que foram replicados nos 5.431 grafos. Podemos observar que 62,3% dos nós especiais replicados são do tipo Zero nos dois terminais e que apenas 0,8% são Zero em um terminal e Down em outro. Na tabela 5.12, podemos ver que 51,9% dos nós normais replicados são filhos dos nós especiais replicados e que em 1,4% dos grafos existem nós normais que são irmãos de nós especiais.

Tabela 5.11: Tipo dos nós especiais que foram replicados no grafos grandes

Tipo	# Nós	% Nós
Up	19	<0,1
Down	0	0,0
Zero	123.305	62,3
Zero Up	67.093	33,9
Zero Down	1.547	0,8
Up Down	6.021	3,0

Tabela 5.12: Relação entre os nós normais e especiais que foram replicados nos grafos grandes

Relação	# Grafos	% Grafos
Pais	86	10,9
Filhos	408	51,9
Irmãos	11	1,4
Pais e filhos	281	35,8

As tabelas 5.13 e 5.14 apresentam os tempos totais, em segundos, que os algoritmos de divisão mínima e corte mínimo com nós especiais demoraram para calcular a divisão nos grafos pequenos e grandes onde houve replicação. Ao analisar a tabela 5.13, podemos ver que o algoritmo de divisão mínima é 0,009% mais lento do que o algoritmo de corte mínimo com nós especiais para estes grafos. Para os grafos grandes, o algoritmo de divisão mínima é 1,67 vezes mais lento do que o de corte mínimo.

Tabela 5.13: Tempos totais de execução dos algoritmos de divisão mínima e corte mínimo com nós especiais para todos os grafos pequenos onde houve replicação

Algoritmo	Tempo (s)
Divisão Mínima com Nós Especiais	20,083
Corte Mínimo com Nós Especiais	20,081

Tabela 5.14: Tempos totais de execução dos algoritmos de divisão mínima e corte mínimo com nós especiais para todos os grafos grandes onde houve replicação

Algoritmo	Tempo (s)
Divisão Mínima com Nós Especiais	50,085
Corte Mínimo com Nós Especiais	30,099

Numa análise mais global, podemos ver que tanto para os grafos pequenos quanto para os grandes, os três algoritmos obtiveram resultados equivalentes entre si em mais de 90% dos casos. Para os grafos onde houve replicação, podemos ver que, tanto para os grafos grandes quanto para os pequenos, os nós utilizados para a replicação foram os nós queridos pelos terminais após o fluxo base em mais de 97% dos casos. Podemos notar também que mais de 77% dos nós replicados são do tipo Zero em pelo menos um dos terminais. Além de que o nó normal replicado ser irmão do nó especial replicado acontece em menos do que 1,5% dos grafos grandes e pequenos.

5.3 Divisão-k Mínima com Nós Especiais

Para avaliar o algoritmo de divisão-k mínima com nós especiais, iremos comparar os seguintes algoritmos:

- O algoritmo de divisão-k mínima com os nós queridos pelo fluxo base (designado pela letra B);
- O algoritmo de divisão-k mínima com os nós queridos pelos fluxo na instância canónica (desiginado pela letra C);
- O algoritmo de corte-k mínimo com nós especiais (designado pela letra A), proposto em [6].

As tabelas 5.15 e 5.16 apresentam os resultados obtidos ao comparar os três algoritmos para os grafos pequenos e grandes, respectivamente. Podemos observar que, para os grafos pequenos, o algoritmo de corte-k mínimo para nós especiais é melhor do que os restantes algoritmos em 7 grafos, enquanto que para os grafos grandes, este mesmo algoritmo é melhor do que os restantes em apenas 1 grafo. Podemos observar também, que o algoritmo de divisão-k mínima com os nós queridos após o fluxo base é melhor que os restantes algoritmos em 2,7% dos grafos pequenos e em 4,8% dos grafos grandes.

Tabela 5.15: Comparação dos algoritmos de divisão-k mínima e corte-k mínimo com nós especiais para os grafos pequenos

Algoritmo	# Grafos	% Grafos
A	7	<0,1
B	2.704	2,7
C	3	<0,1
AB	83	<0,1
BC	8	<0,1
AC	7.880	7,8
ABC	89.625	89,4

Tabela 5.16: Comparação dos algoritmos de divisão-k mínima e corte-k mínimo com nós especiais para os grafos grandes

Algoritmo	# Grafos	% Grafos
A	1	<0,1
B	950	4,8
C	0	0,0
AB	1	<0,1
BC	0	0,0
AC	5.223	26,1
ABC	13.825	69,1

As tabelas 5.17 e 5.18 apresentam os tempos totais, em segundos, que os algoritmos de divisão-k mínima (listagem 4.9) e corte-k mínimo com nós especiais demoraram para calcular a divisão-k para todos os grafos pequenos e grandes, respectivamente. Ao analisar a tabela 5.17, notamos que o algoritmo de divisão-k mínima com nós especiais é 0,11% mais lento do que o algoritmo de corte-k mínimo. Enquanto que para os grafos grandes, o algoritmo de divisão-k mínima com nós especiais é 1,02 vezes mais lento do que o de corte-k mínimo com nós especiais.

Tabela 5.17: Tempos totais de execução dos algoritmos de divisão-k mínima e corte-k mínimo com nós especiais para todos os grafos pequenos

Algoritmo	Tempo (s)
Divisão-k Mínima com Nós Especiais	30,217
Corte-k Mínimo com Nós Especiais	30,183

Tabela 5.18: Tempos totais de execução dos algoritmos de divisão-k mínima e corte-k mínimo com nós especiais para todos os grafos grandes

Algoritmo	Tempo (s)
Divisão-k Mínima com Nós Especiais	880,735
Corte-k Mínimo com Nós Especiais	861,328

Análise dos Grafos com Replicação

Ao correr o algoritmo de divisão-k mínima com nós especiais para os grafos pequenos, obtivemos 1.175 grafos replicados. A seguir apresentamos a tabela 5.19, onde comparamos os algoritmos B e C apenas nestes grafos. Ao analisar esta tabela, podemos ver que em 99,7% dos grafos foram utilizados os nós queridos pelos terminais após o fluxo base. E em apenas 3 grafos foram utlizados os nós queridos pelos terminais após o fluxo na instância canónica.

Tabela 5.19: Comparação dos algoritmos de divisão-k mínima para os grafos pequenos onde houve replicação

Algoritmo	# Grafos	% Grafos
B	1.164	99,0
C	3	0,3
BC	8	0,7

Dos 1.175 grafos pequenos, 1.013 grafos replicaram apenas nós especiais e 162 grafos replicaram nós especiais e normais. A seguir apresentamos as tabelas 5.20 e 5.21 que indicam os tipos dos nós especiais replicados e a relação entre os nós normais e especiais que foram replicados. Podemos observar que nenhum nó especial replicado é Down nos dois terminais, enquanto que 858 são Up nos dois terminais. Podemos notar também que não existem grafos onde os nós normais replicados são filhos dos nós especiais e em apenas 1 grafo os nós normais são irmãos do nó especial.

Tabela 5.20: Tipo dos nós especiais que foram replicados nos grafos pequenos

Tipo	# Nós	% Nós
Up	858	61,4
Down	0	0,0
Zero	201	14,4
Zero Up	325	23,2
Zero Down	1	<0,1
Up Down	13	0,9

Tabela 5.21: Relação entre os nós normais e especiais que foram replicados nos grafos pequenos

Relação	# Grafos	% Grafos
Pais	124	76,5
Filhos	0	0,0
Irmãos	1	0,6
Pais e filhos	37	22,8

Para os grafos grandes, tivemos que em 607 grafos houve replicação. A tabela 5.22 apresenta a comparação dos algoritmos B e C nestes grafos. Ao analisar esta tabela, podemos ver que todos os grafos onde houve replicação utilizaram os nós queridos pelos

terminais após o fluxo base.

Tabela 5.22: Comparação dos algoritmos de divisão-k mínima para os grafos grandes onde houve replicação

Algoritmo	# Grafos	% Grafos
B	607	100
C	0	0,0
BC	0	0,0

Dos 607 grafos onde houve replicação, 576 replicaram apenas nós especiais e 31 replicaram nós normais e especiais. A seguir apresentamos as tabelas 5.23 e 5.24 que indicam os tipos dos nós especiais e a relação entre os nós normais e especiais que foram replicados. Ao analisar a tabela 5.23, notamos que nenhum nó replicado é Down nos dois terminais, enquanto que 10.733 nós são Zero nos dois terminais. Ao observar a tabela 5.24, notamos que em 96,8% dos grafos que replicam nós normais e especiais, os nós normais são filhos dos nós especiais. E em apenas 1 grafo os nós normais replicados são irmãos dos nós especiais replicados.

Tabela 5.23: Tipo dos nós especiais que foram replicados nos grafos grandes

Tipo	# Nós	% Nós
Up	42	0,2
Down	0	0,0
Zero	10.733	59,9
Zero Up	6.577	36,7
Zero Down	126	0,7
Up Down	423	2,4

Tabela 5.24: Relação entre os nós normais e especiais que foram replicados nos grafos grandes

Relação	# Grafos	% Grafos
Pais	0	0,0
Filhos	30	96,8
Irmãos	1	3,2
Pais e filhos	0	0,0

As tabelas 5.25 e 5.26 apresentam os tempos totais, em segundos, que os algoritmos de divisão-k mínima e corte-k mínimo com nós especiais demoraram para calcular a divisão-k nos grafos pequenos e grandes, respectivamente. Ao analisar a tabela 5.25, podemos ver que o algoritmo de divisão-k mínima é 0,005% mais lento do que o algoritmo de corte-k mínimo. Enquanto que para os grafos grandes, o algoritmo de divisão-k mínima é 1,007 vezes mais lento do que o de corte-k mínimo.

Tabela 5.25: Tempos totais de execução dos algoritmos de divisão-k mínima e corte-k mínimo com nós especiais para todos os grafos pequenos onde houve replicação

Algoritmo	Tempo (s)
Divisão-k Mínima com Nós Especiais	20,070
Corte-k Mínimo com Nós Especiais	20,069

Tabela 5.26: Tempos totais de execução dos algoritmos de divisão-k mínima e corte-k mínimo com nós especiais para todos os grafos grandes onde houve replicação

Algoritmo	Tempo (s)
Divisão-k Mínima com Nós Especiais	40,351
Corte-k Mínimo com Nós Especiais	40,084

Numa análise mais global, podemos ver que para os grafos pequenos os três algoritmos obtiveram resultados equivalentes entre si em 89,4% dos casos, enquanto que para os grafos grandes, os mesmos algoritmos obtiveram o melhor resultado em apenas 69,1% dos casos. Para os grafos onde houve replicação, podemos notar que, tanto para os grafos pequenos quanto para os grandes, foram utilizados os nós queridos pelos terminais após o fluxo base em mais de 99% dos casos. Também podemos ver que menos de 3,2% dos nós replicados são do tipo Down em pelo menos um dos terminais. E que assim como com o algoritmo de divisão mínima, em menos do que 3,3% dos grafos o nó normal replicado é irmão do nó especial replicado.

Capítulo 6

Conclusões e Trabalho Futuro

Neste trabalho foram propostos os algoritmos de divisão mínima e divisão-k mínima com nós especiais que têm como objetivo tentar resolver os conflitos existentes entre terminais através da replicação e, consequentemente, minimizar o tráfego de dados entre os *data centers*.

Para o algoritmo de divisão mínima com nós especiais, foram criadas versões ao replicarmos apenas os nós queridos pelo fluxo base e apenas os nós queridos após o fluxo na instância canónica. Considerando que os nós queridos pelo fluxo base e após o fluxo na instância canónica teriam que ser sempre calculados e que algumas vezes conseguimos melhorar a capacidade da divisão ao replicar com os nós do fluxo base e outras vezes com os nós após o fluxo na instância canónica, então criámos o algoritmo apresentado neste documento, que considera os dois casos, o algoritmo de corte mínimo com nós especiais e retorna sempre o melhor caso. Uma vez que este algoritmo com dois *data centers* apresentou resultados satisfatórios, usámos a mesma abordagem para o algoritmo com k *data centers* (terminais). O algoritmo de divisão-k mínima com nós especiais proposto neste documento considera o caso de replicar os nós queridos pelo fluxo base, os nós queridos após o fluxo na instância canónica e a possibilidade de não haver replicação (corte-k mínimo com nós especiais).

Ao analisarmos os resultados experimentais, podemos concluir que ambos os algoritmos apresentam bons resultados, apenas não encontrando a melhor solução, de entre as soluções encontradas pelos algoritmos comparados, em menos de 0,1% dos casos. Podemos concluir também que em alguns casos, que variam entre 0,12% e 3,98%, a replicação irá minimizar a transferência de dados entre *data centers*, tanto para dois *data centers* como para k *data centers*. Nestes grafos, foram utilizados os nós queridos pelos terminais após o fluxo base em mais de 97% dos casos para ambos os algoritmos. Também podemos ver que os nós especiais replicados são, em mais de 99% dos casos, do tipo Zero ou Up

em pelo menos um dos terminais.

Como visto anteriormente, existem casos onde a replicação melhora a capacidade da divisão para ambos os algoritmos. Então, como trabalho futuro, seria interessante a criação de um algoritmo que, além de calcular se a replicação irá melhorar a capacidade da divisão, também considerasse o custo da replicação para os *data centers* e assim decidir se deve ou não haver replicação de nós.

Outro trabalho futuro interessante seria o estudo da relação entre os resultados obtidos pelos algoritmos propostos e a melhor solução e assim determinar o erro máximo de aproximação dos algoritmos propostos.

Bibliografia

[1] B. Hopkins e R. Wilson. "The Truth about Konigsberg". Em: *The College Mathematics Journal* 35 (mai. de 2004), pp. 198–207.

[2] G. Gutin e A. P. Punnen, eds. *The Traveling Salesman Problem and Its Variations*. Boston, MA: Springer US, 2007. ISBN: 978-0-306-48213-7. DOI: 10.1007/0-306-48213-4_14. URL: https://doi.org/10.1007/0-306-48213-4_14.

[3] P. Franklin. "The Four Color Problem". Em: *American Journal of Mathematics* 44.3 (1922), pp. 225–236. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2370527.

[4] S. Ji e B. Li. "Wide area analytics for geographically distributed datacenters". Em: *Tsinghua Science and Technology* 21.2 (2016), pp. 125–135. DOI: 10.1109/TST.2016.7442496.

[5] K. Kloudas, M. Mamede, N. Preguiça e R. Rodrigues. "Pixida: Optimizing Data Parallel Jobs in Wide-area Data Analytics". Em: *Proc. VLDB Endow.* 9.2 (out. de 2015), pp. 72–83. ISSN: 2150-8097. DOI: 10.14778/2850578.2850582. URL: http://dx.doi.org/10.14778/2850578.2850582.

[6] R. C. Martins. "Cortes Mínimos em Grafos". Tese de mestrado. Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2016.

[7] L. Chen, S. Liu, B. Li e B. Li. "Scheduling jobs across geo-distributed datacenters with max-min fairness". Em: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8056949.

[8] M. W. Convolbo, J. Chou, C.-H. Hsu e Y. C. Chung. "GEODIS: towards the optimization of data locality-aware job scheduling in geo-distributed data centers". Em: *Computing* (2017). ISSN: 1436-5057. DOI: 10.1007/s00607-017-0564-7. URL: https://doi.org/10.1007/s00607-017-0564-7.

[9] W. Xiao, W. Bao, X. Zhu e L. Liu. "Cost-Aware Big Data Processing Across Geo-Distributed Datacenters". Em: *IEEE Transactions on Parallel and Distributed Systems* 28.11 (2017), pp. 3114–3127. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2708120.

[10] K. Oh, A. Chandra e J. Weissman. "TripS: Automated Multi-tiered Data Placement in a Geo-distributed Cloud Environment". Em: *Proceedings of the 10th ACM International Systems and Storage Conference*. SYSTOR '17. Haifa, Israel: ACM, 2017, 12:1–12:11. ISBN: 978-1-4503-5035-8. DOI: 10.1145/3078468.3078485. URL: http://doi.acm.org/10.1145/3078468.3078485.

[11] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker e I. Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". Em: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737.

[12] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom e S. Whittle. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". Em: *Very Large Data Bases*. 2013, pp. 734–746.

[13] A. Vulimiri, C. Curino, B. Godfrey, T. Jungblut, J. Padhye e G. Varghese. "Global analytics in the face of bandwidth and regulatory constraints". Em: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*. USENIX, 2015, pp. 323–336.

[14] A. Rabkin, M. Arye, S. Sen, V. S. Pai e M. J. Freedman. "Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area". Em: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 275–288. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/rabkin.

[15] T. H. Cormen, C. Stein, R. L. Rivest e C. E. Leiserson. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[16] L. R. Ford e D. R. Fulkerson. "Maximal Flow Through a Network". Em: *Classic Papers in Combinatorics*. Ed. por I. Gessel e G.-C. Rota. Boston, MA: Birkhäuser Boston, 1987, pp. 243–248. ISBN: 978-0-8176-4842-8. DOI: 10.1007/978-0-8176-4842-8_15. URL: https://doi.org/10.1007/978-0-8176-4842-8_15.

[17] J. Edmonds e R. M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". Em: *J. ACM* 19.2 (abr. de 1972), pp. 248–264. ISSN: 0004-5411. DOI: 10.1145/321694.321699. URL: http://doi.acm.org/10.1145/321694.321699.

[18] Y. Dinitz. "Dinitz' Algorithm: The Original Version and Even's Version". Em: *Theoretical Computer Science: Essays in Memory of Shimon Even*. Ed. por O. Goldreich, A. L. Rosenberg e A. L. Selman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 218–240. ISBN: 978-3-540-32881-0. DOI: 10.1007/11685654_10. URL: https://doi.org/10.1007/11685654_10.

[19] A. V. Goldberg e R. E. Tarjan. "A New Approach to the Maximum-flow Problem". Em: *J. ACM* 35.4 (out. de 1988), pp. 921–940. ISSN: 0004-5411. DOI: 10.1145/48014.61051. URL: http://doi.acm.org/10.1145/48014.61051.

[20] R. Cerulli, M. Gentili e A. Iossa. "Efficient preflow push algorithms". Em: *Computers & Operations Research* 35.8 (2008). Queues in Practice, pp. 2694 –2708. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2006.12.023. URL: http://www.sciencedirect.com/science/article/pii/S030505480600325X.

[21] J. B. Orlin. "Max Flows in O(Nm) Time, or Better". Em: *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*. STOC '13. Palo Alto, California, USA: ACM, 2013, pp. 765–774. ISBN: 978-1-4503-2029-0. DOI: 10.1145/2488608.2488705. URL: http://doi.acm.org/10.1145/2488608.2488705.

[22] V. Malhotra, M. Kumar e S. Maheshwari. "An O(|V|3) algorithm for finding maximum flows in networks". Em: *Information Processing Letters* 7.6 (1978), pp. 277 –278. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(78)90016-9. URL: http://www.sciencedirect.com/science/article/pii/0020019078900169.

[23] A. V. Goldberg e S. Rao. "Beyond the Flow Decomposition Barrier". Em: *J. ACM* 45.5 (set. de 1998), pp. 783–797. ISSN: 0004-5411. DOI: 10.1145/290179.290181. URL: http://doi.acm.org/10.1145/290179.290181.

[24] N. Guttmann-Beck e R. Hassin. "Approximation Algorithms for Minimum K -Cut". Em: *Algorithmica* 27.2 (2000), pp. 198–207. ISSN: 1432-0541. DOI: 10.1007/s004530010013. URL: https://doi.org/10.1007/s004530010013.

[25] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour e M. Yannakakis. "The Complexity of Multiterminal Cuts". Em: *SIAM J. Comput.* 23.4 (ago. de 1994), pp. 864–894. ISSN: 0097-5397. DOI: 10.1137/S0097539792225297. URL: http://dx.doi.org/10.1137/S0097539792225297.

[26] K. Kloudas, M. Mamede, N. Preguiça e R. Rodrigues. *The Generalized Min-Cut Problem*. Rel. téc. NOVA LINCS, 2015. URL: http://ctp.di.fct.unl.pt/~mm/techreport-2015-09.pdf.